UNIVERSITÉ DE MONTRÉAL

RÉSOLUTION DE PROBLÈME PAR SUIVI DE MÉTRIQUES DANS LES SYSTÈMES VIRTUALISÉS

JULIEN DESFOSSEZ
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION DU
DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
DÉCEMBRE 2011

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

RÉSOLUTION DE PROBLÈME PAR SUIVI DE MÉTRIQUES DANS LES SYSTÈMES VIRTUALISÉS

présenté par : DESFOSSEZ Julien
en vue de l'obtention du diplôme de : Maîtrise ès Sciences Appliquées
a été dûment accepté par le jury constitué de :

Mme. BELLAÏCHE Martine, Ph.D., présidente
M. DAGENAIS Michel, Ph.D., membre et directeur de recherche
M. BOYER François-Raymond, Ph.D., membre

À tous ceux qui ont cru en moi,
et aux autres...

REMERCIEMENTS

Je tiens à remercier mon directeur de recherche Michel Dagenais pour sa disponibilité, son écoute et son intérêt infini pour la recherche de la connaissance et de la compréhension. Je tiens également à remercier Révolution Linux et particulièrement Benoit des Ligneris pour la confiance, le soutien moral et financier, et les encouragements qu'ils m'ont apporté pour mener à bien cette recherche.

De plus, je tiens à reconnaître le soutien financier offert par l'organisme MITACS ainsi que la contribution du Fonds Québécois de Recherche Nature et Technologie (FQRNT).

Enfin je tiens à remercier mes collègues du laboratoire DORSAL avec qui les nombreuses discussions techniques ont fait mûrir les réflexions présentées dans ce mémoire. Un remerciement particulier à Mathieu Desnoyers pour sa patience et son acharnement à toujours sélectionner la solution optimale et irréfutable.

RÉSUMÉ

Le traçage en espace noyau et utilisateur est un sujet de plus en plus populaire étant donné le niveau de détail et la précision des données qu'il permet d'extraire sur les activités d'un système d'exploitation. Bien souvent par contre, les données recueillies sont tellement détaillées et complexes qu'elles requièrent beaucoup de temps et d'expertise à analyser et à comprendre. C'est pourquoi les traceurs sont habituellement utilisés par les développeurs et plus rarement par les administrateurs systèmes ou les utilisateurs. Cependant, les données recueillies, lorsque bien traitées et présentées, peuvent apporter des informations très pertinentes pour une grande variété d'utilisateurs à un coût très faible. De plus, la virtualisation déjà très présente en entreprise apporte de nouveaux défis dans l'enregistrement de traces et il est difficile avec les méthodologies actuelles de travailler efficacement avec de telles traces. Enfin, contrairement aux environnements de développement, les environnements de production tels que les centres de données sont souvent composés de serveurs fonctionnant sur différentes versions de système d'exploitation, utilisant des logiciels et technologies variées, c'est pourquoi il est nécessaire d'apporter une méthodologie générique afin de régler le plus efficacement possible les différents problèmes pouvant survenir dans un maximum de cas d'utilisation.

L'objectif de cette recherche est de montrer qu'il est possible d'utiliser les informations recueillies par des traceurs pour combler l'écart présent entre les outils de diagnostic pour administrateurs systèmes et pour développeurs.

Pour ce faire, nous allons étudier les méthodes actuelles d'analyse et de résolution de problèmes dans les environnements de production, ensuite nous verrons comment faire efficacement le lien entre des événements enregistrés au niveau utilisateur, dans le noyau et dans l'hyperviseur KVM, enfin nous présenterons une nouvelle solution développée pour rendre l'utilisation de traces noyau pratique pour diagnostiquer des problèmes complexes dans des environnements réels sans nécessiter de bagage en développement noyau.

Tout au long de cette recherche, la méthode expérimentale consiste en la corrélation entre l'activité du système et l'abstraction faite par les outils d'analyse basés sur les informations de trace. En fonction de la problèmatique traitée, différents scénarios seront créés sur les systèmes en test. Le succès d'une expérience sera jugé selon la capacité à automatiser avec précision la synthèse et l'analyse des informations, de telle sorte qu'un utilisateur puisse rapidement profiter des données produites pour comprendre des problèmes complexes apparaissant sur les systèmes d'exploitation.

L'hypothèse servant de point de départ de ce travail est qu'il est possible d'utiliser les informations produites par un traceur pour diagnostiquer des problèmes à différents niveaux (utilisateur, noyau, hyperviseur) sans perturber le système.

Les résultats de ce travail sont la création et l'intégration aux outils existants d'un mécanisme pour faire correspondre des traces enregistrées à différents niveaux d'un système d'exploitation avec un impact minimal, ce qui permet de comprendre et mesurer les liens entre les différentes couches. De plus, de nouvelles métriques et une méthodologie pour rendre la lecture de traces reliées à la virtualisation sont développées et testées afin de rendre clair les interactions entre une machine virtuelle et le système hôte. Enfin, cette recherche introduit également un nouvel outil permettant aux administrateurs systèmes d'utiliser des traces provenant du noyau selon leur méthodologie de travail habituelle.

Le résultat final est que l'utilisation du traçage est viable pour la supervision d'un parc informatique. Les bibliothèques et outils développés servent de base à l'intégration avec les systèmes de supervision traditionnels, permettant la réaction rapide aux éventuels problèmes.

ABSTRACT

Tracing in user or kernel-space is becoming a popular subject. It provides a lot of information about the operating system with high precision. One problem with tracing though, is the fact that it is designed for developpers, and the amount of information and the level of detail is often too much for a system administrator or a user. However, if processed and presented correctly, the data produced by tracers can be of great use for a wide variety of users. Moreover, virtualisation increasingly used in enterprise, brings new challenges to tracing and current methodologies are not enough to record and work with such traces. Data centers are often running a lot of servers with different versions of software and operating systems for various reasons. It is therefore important to develop new generic methodologies covering the most use cases possible, to be able to quickly identify and solve the problems happening on those systems with a reliable data source such as tracing.

The objective of this research is to prove that we can use the data produced by a tracer to bridge the gap between tools in use by system administrators and developpers.

In order to do that, we will study current methodologies for identifying and solving problems in production environments, then we will look at a way to synchronise traces recorded in the kernel, in user-space and in the hypervisor, and finally we will present a new way to process tracing data to assist in the resolution of complex problems in a real production environment, without requiring knowledge in kernel programming.

The hypothesis serving as starting point for this work is that it is possible to use the information produced by a tracer to identify problems happening at various levels (user, kernel, hypervisor) without disrupting the system.

The results of this work are the creation and integration in existing tools of a method to synchronise traces recorded at various levels on the same machine with a minimal impact. Moreover, new metrics and methodologies have been developped and tested to assist in the analysis process of traces related to virtualisation. Finally, this research introduces a new tool that allows system administrators to use kernel traces with their usual work habits.

The final result is that using tracing is a valid and good option to monitor a data center. Thus, the libraries and tools developped are becoming the basis of the integration with the usual monitoring services, which allow a quick response in case of a problem.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES ANNEXES

LISTE DES SIGLES ET ABRÉVIATIONS

CPU	Central Processing Unit
CTF	Common Trace Format
ETW	Event Tracing for Windows
LTTV	Linux Trace Toolkit Viewer
LTTng	Linux Trace Toolkit Next Generation
PID	Process Identifier
PPID	Parent Process Identifier
TID	Thread Identifier
TSC	Time Stamp Counter
UST	User-Space Tracer
VFS	Virtual File System
VM	Virtual Machine
vDSO	Virtual Dynamically-linked Shared Object

Chapitre 1

INTRODUCTION

1.1 Problème étudié et buts poursuivis

Un parc informatique est un ensemble d'ordinateurs et d'équipements de communication fonctionnant ensemble afin d'assurer un service. Les équipements, logiciels et services fournis, et les intéractions entre ceux-ci, varient beaucoup selon le parc. De plus en plus, les parcs informatiques d'envergure intègrent une part de machines virtuelles pour la grande flexibilité qu'elles apportent en terme de gestion et de sécurité. Ces environnements n'ont jamais été la cible première des développeurs de solution de traçage. D'habitude, un traceur, qu'il soit en espace noyau ou utilisateur, est développé pour assister les développeurs dans la compréhension de problèmes complexes. L'information extraite par ces logiciels est d'une grande précision et contient beaucoup de données qui sont aussi pertinentes pour les administrateurs systèmes faisant face à des problèmes difficiles à résoudre avec leurs outils habituels. Le problème est de prendre cette source d'information brute, la compléter avec les éléments manquants tels que les informations liées à la virtualisation, puis de la transformer afin de produire un résultat satisfaisant pour la gestion d'un parc informatique hétérogène, comprenant ou non une composante de virtualisation. L'objectif final est d'obtenir, par des métriques fiables, une méthodologie permettant de faciliter le diagnostic de problèmes complexes.

1.2 Définitions et concepts de base

1.2.1 Traçage

Le traçage, qu'il soit lié à une application ou dans le noyau, consiste à enregistrer des événements durant l'exécution du système. Habituellement utilisé comme outil de développement, il sert à obtenir des informations sur le comportement d'un système en minimisant l'impact de cette prise d'information sur celui-ci. Les événements sont habituellement constitués d'une estampille temporelle, d'un identificateur, optionnellement de paramètres liés au type d'événement et d'informations plus génériques telles que le numéro de processeur.

Il est habituellement possible pour l'utilisateur de sélectionner les points de trace qu'il désire activer afin d'enregistrer seulement l'information qui l'intéresse, et par la même occasion limiter la charge supplémentaire imposée par le traceur.

1.2.2 Virtualisation

La virtualisation est un domaine qui existe depuis de nombreuses années. De nombreuses technologies se sont succédées et la recherche est encore très active sur ce sujet. L'objectif de virtualiser une machine est de séparer le lien physique qui relie un système d'exploitation à un ordinateur. Ainsi, il est possible de déplacer une machine virtuelle sur du matériel différent, ou exécuter plusieurs machines virtuelles sur la même machine physique. À l'heure actuelle, on utilise les machines virtuelles pour faciliter la gestion. D'un point de vue physique, on peut voir une machine virtuelle comme un programme s'exécutant sur un serveur.

Dans le cadre de cette recherche, l'accent est mis sur la technologie courante la plus employée qui consiste à un hyperviseur fonctionnant dans l'espace noyau de la machine hôte, responsable d'attribuer les ressources physiques aux machine virtuelles, et de les laisser exécuter nativement leurs instructions non-privilégiées en profitant de la technologie Intel VT ou AMD HVM.

1.2.3 Types de problèmes à résoudre

Tout au long de ce mémoire nous parlons des problèmes complexes que nous cherchons à résoudre, nous appelons problèmes complexes des problèmes qui impliquent une interaction difficile à comprendre avec les outils habituels et qui concernent différents composants logiciels ou matériels. Plus concrètement, ce peut être des problèmes de latence impliquant une mauvaise gestion de la mémoire tampon, ou des problèmes difficilement reproductibles qui nécessitent un certain nombre de paramètres pour survenir.

1.3 Démarche de l'ensemble du travail

Le travail présenté est découpé en deux axes distincts : tout d'abord la capture et la synchronisation *a posteriori* de traces de l'hyperviseur et de l'espace utilisateur afin d'avoir une vue globale des opérations liées à la virtualisation exécutées sur la machine hôte, puis l'analyse automatisée et le traitement de ces traces pour produire des métriques et de l'information pertinente destinée à des administrateurs systèmes.

1.4 Organisation générale du document

Le chapitre 2 présente une revue de littérature concernant le traçage, la virtualisation et l'automatisation de la recherche d'erreurs sur un système d'exploitation. Dans le chapitre 3, nous verrons en détail les métriques utilisées et les concepts abordés lors de ce mémoire. Ensuite, le chapitre 4 intègre le contenu de l'article «Efficiently Tracing Across The Hypervisor Layers» qui traite du travail réalisé pour tracer un hyperviseur, représenter les métriques pertinentes sur l'utilisation du processeur virtuel ainsi que de la synchronisation des traces collectées indépendemment en espace utilisateur et noyau. Dans le chapitre 5, l'article «Linux Kernel Tracing as a Problem-finding Methodology» est intégré, celui-ci traite du défi associé à l'utilisation de traces pour un administrateur de parc informatique et présente une méthodologie pour faire le lien entre le domaine de la collecte de traces et celui de la recherche de problèmes systèmes. Le chapite 6 est une discussion générale sur les résultats présentés dans ce mémoire et apporte des résultats complémentaires aux deux articles. Plus particulièrement il présente l'outil `LTTngTop` développé dans le cadre de cette recherche. Enfin le chapitre 7 conclut ce mémoire avec une synthèse critique et des pistes pour des recherches futures.

Chapitre 2

REVUE DE LITTÉRATURE

Ce chapitre présente l'état de l'art du domaine de la capture de traces noyau et en espace utilisateur sous Linux. Nous étudions également les technologies de virtualisation et le traçage de ces solutions.

2.1 Traçage sous Linux

Le traçage est une technique permettant de récupérer un maximum d'information sur une application en cours d'exécution avec un minimum d'impact sur celle-ci. Contrairement au déverminage, l'objectif du traçage est d'obtenir des informations sans interrompre l'exécution du programme cible. De plus, le traçage vise à donner une vue d'ensemble des intéractions entre les composants du système d'exploitation, c'est pourquoi le traçage noyau se concentre sur la possibilité d'offrir une vue globale de l'activité du système.

2.1.1 Traçage en espace noyau

Instrumentation statique avec `TRACE_EVENT()`

De nombreuses technologies se sont succédées sous Linux pour arriver à ce que nous appelons maintenant l'état de l'art. Après ces nombreuses années à justifier l'importance du traçage dans le noyau Linux et à s'accorder sur un modèle viable pour les différentes approches, la macro `TRACE_EVENT` (Rostedt, 2010) est devenue la base et le standard des points d'instrumentation statiques dans le noyau. Cette macro permet aux développeurs de créer rapidement et de manière très propre des emplacements dans leur code sur lesquels ils peuvent connecter le traceur de leur choix pour récupérer de l'information. Cette macro ne dépend pas d'un traceur en particulier et à l'heure actuelle, `ftrace`, `LTTng`, `perf` et `SystemTap` l'utilisent comme source d'information.

Instrumentation dynamique avec `kprobes`

À l'opposé des points d'instrumentation statiques qui nécessitent de modifier et de re-compiler le code source, `kprobes` (Rostedt, 2005) est un mécanisme permettant l'insertion

dynamique de points de trace dans le noyau. Il s'agit d'insérer, à l'endroit désiré dans l'espace noyau, un mécanisme équivalent à un point d'arrêt ainsi que le code pour le traiter automatiquement. Cette méthode d'instrumentation ayant un coût plus élevé que l'instrumentation statique, elle est recommandée pour de l'analyse *ad hoc*. Étant donné que le code responsable de traiter ces points de trace dynamique s'exécute dans le noyau, il est possible d'utiliser cette méthode à d'autres fins que le traçage, par exemple l'injection de fautes arbitraires.

Le code présent à l'annexe B est un exemple d'utilisation de `kprobes`. Ici un module noyau est créé pour compter le nombre de fois que la fonction du noyau `generic_make_request` est appelée. Lorsque le module est chargé, le code est inséré, et lorsqu'il est déchargé le compteur s'affiche. Ainsi, un affichage similaire à celui-ci est présenté :

```
[146149.728152] kprobe registered
[146159.796184] kprobe unregistered
[146159.796194] generic_make_request() called 390 times.
```

Donc on peut voir que pendant les 10 secondes où le code a été exécuté, la fonction qui nous intéresse a été appelée 390 fois. Ce mécanisme permet d'instrumenter rapidement le noyau sans nécessiter de le recompiler. Toutefois, il est nécessaire de bien connaître le code du noyau pour tirer partie de cette solution.

Le traceur `ftrace`

Le traceur `ftrace` (Edge, 2009a) a débuté comme une initiative sur le noyau Linux temps réel mais depuis a été ramené dans le noyau Linux officiel. L'objectif initial était d'avoir un mécanisme automatisé pour connaître l'ordre des fonctions exécutées dans le noyau afin de connaître le chemin critique pendant une certaine période de temps. Depuis, ce traceur a beaucoup évolué et possède maintenant des modules pour faire des analyses plus poussées telles que des analyses de latence, des analyses sur le temps où les interruptions sont désactivées, des analyses sur l'ordonnanceur. De plus, celui-ci utilise la macro `TRACE_EVENT` pour extraire des données sur le code exécuté dans le noyau. Un avantage de `ftrace` est le fait qu'il est entièrement intégré au noyau et aucun outil externe n'est nécessaire pour le contrôler ou même lire les données. Tout le contrôle se passe dans le pseudo système de fichier `debugfs` et les fichiers de trace de base sont au format texte. Étant donné le volume de données généré, il est également possible de sortir la trace dans un format binaire et utiliser l'outil graphique `KernelShark` pour consulter plus en détail les informations recueillies.

L'exemple ci-dessous est extrait d'une trace enregistrée par `ftrace`. On y voit des opérations de gestion de mémoire du noyau, d'événements d'ordonnanceur (un changement de contexte) et d'appels systèmes qui s'exécutent sur le même processeur (001).

```
/usr/bin/x-term-7503  [001]  2478.885352: kmem_cache_free:
    call_site=ffffffffa007ed38 ptr=ffff8800b186f318
/usr/bin/x-term-7503  [001]  2478.885353: timer_start:
    timer=ffff88012ded36c0 function=wakeup_timer_fn
    expires=4317208064 [timeout=150295]
/usr/bin/x-term-7503  [001]  2478.885361: kmem_cache_free:
    call_site=ffffffffa007ed38 ptr=ffff8800b186f318
/usr/bin/x-term-7503  [001]  2478.885369: sched_wakeup:
    comm=evince pid=27723 prio=120 success=1 target_cpu=001
/usr/bin/x-term-7503  [001]  2478.885369: sched_stat_runtime:
    comm=/usr/bin/x-term pid=7503 runtime=63492 [ns]
    vruntime=716792871 [ns]
/usr/bin/x-term-7503  [001]  2478.885371: sched_switch:
    prev_comm=/usr/bin/x-term prev_pid=7503 prev_prio=120
    prev_state=R ==> next_comm=evince next_pid=27723 next_prio=120
evince-27723 [001]  2478.885377: sys_exit: NR 7 = 1
evince-27723 [001]  2478.885384: sys_enter: NR 16
    (e, 541b, 7fffaa8b61fc, 1, cde188, 6c4b)
evince-27723 [001]  2478.885385: sys_exit: NR 16 = 0
```

Cette information est intéressante pour les développeurs qui souhaitent connaître ce qui s'est passé exactement sur le système ainsi que le temps pris par chaque opération.

Le traceur `perf`

Le traceur `perf` (Edge, 2009b) est un autre traceur intégré au noyau Linux. Celui-ci a été conçu à l'origine pour fournir une interface facile d'accès vers les compteurs de performance présents dans les processeurs. Depuis, il a évolué pour également s'interfacer avec la macro `TRACE_EVENT` et fournir le même type de données Ainsi, il est possible de l'utiliser pour produire rapidement des statistiques sur le nombre de fois qu'un point d'instrumentation a été exécuté, ou le nombre d'événements enregistrés sur un processeur pendant une période donnée. Le traceur se contrôle par l'outil du même nom situé dans le répertoire `tools/` des sources du noyau.

Dans sa forme la plus simple, des statistiques comme celles présentées ci-dessous sont générés :

```
./perf stat ls >/dev/null
```

```
Performance counter stats for 'ls':
2.837672  task-clock       0.834 CPUs utilized
0         context-switches 0.000 M/sec
0         CPU-migrations   0.000 M/sec
263       page-faults      0.093 M/sec
1,646,666 cycles           0.580 GHz
1,682,709 instructions     1.02  insns per cycle
342,848   branches         120.820 M/sec
15,159    branch-misses    4.42% of all branches [26.66%]

0.003400852 seconds time elapsed
```

Ces statistiques proviennent d'informations sorties des compteurs de performances matériels et logiciels.

De plus, `perf` peut être utilisé pour afficher les fonctions les plus fréquemment utilisées dans le noyau et même rentrer dans l'assembleur de chaque fonction. Le document Gorman (2009) issu de la documentation du noyau Linux donne une excellente méthode pour déterminer la source d'un problème avec `perf` et les points d'instrumentation dans le cas où celle-ci se manifeste par une utilisation intensive d'un processeur. Cette analyse descend jusqu'à identifier l'instruction à l'intérieur de la fonction fautive dans le programme.

Toutefois, les informations extraites de `perf` sont des valeurs échantillonnées. À chaque fois que la valeur limite fixée par le traceur est atteinte, une interruption est générée et l'information est enregistrée. Ceci a comme avantage de limiter l'impact du traceur sur le système, par contre la précision des données générées en souffre.

Le traceur `LTTng`

Le traceur `LTTng` (M. Desnoyers, 2006) est le premier traceur a avoir pris en considération l'importance de l'impact sur le système (Desnoyers, 2009). Depuis ses débuts en 2006, son développement a toujours progressé à l'extérieur des sources du noyau officiel. `LTTng` est un ensemble d'outils en espace noyau et utilisateur permettant de générer, contrôler et analyser des traces. Avant la version 2.0, `LTTng` était un ensemble de modifications à appliquer sur le noyau officiel et apportait beaucoup d'instrumentation personnalisée. Avec la version 2.0, tout le coeur du traceur fonctionne sous la forme de module et il utilise comme source de données l'instrumentation statique fournie par les `TRACE_EVENT`, l'instrumentation dynamique fournie par `kprobes`, le traçage de fonction et la capture des compteurs de performance. Les traces générées dans la version 2.0 sont au format `Common Trace Format`

(Desnoyers, 2011).

L'outil `Babeltrace` est utilisé pour lire et convertir les traces stockées dans ce format binaire vers du texte. Par exemple, l'affichage suivant est un extrait d'une trace :

```
[146860151723787] sched_switch: { 0 }, { prev_comm = "ltt-sessiond",
  prev_tid = 17380, prev_prio = 20, prev_state = 1,
  next_comm = "kworker/0:1", next_tid = 16454, next_prio = 20 }
[146860151747882] sched_switch: { 0 }, { prev_comm = "kworker/0:1",
  prev_tid = 16454, prev_prio = 20, prev_state = 1,
  next_comm = "/usr/bin/x-term", next_tid = 17099, next_prio = 20 }
[146860151765761] exit_syscall: { 0 }, { ret = 1 }
[146860151786504] sys_read: { 0 }, { fd = 16, buf = 0x27EE4A4,
  count = 7000 }
[146860151795863] exit_syscall: { 0 }, { ret = 1145 }
[146860151798796] sys_read: { 0 }, { fd = 16, buf = 0x27EE91D,
  count = 5855 }
[146860151803196] exit_syscall: { 0 }, { ret = -11 }
[146860151814231] sys_read: { 0 }, { fd = 4, buf = 0x20A0AB4,
  count = 4096 }
[146860151819399] exit_syscall: { 0 }, { ret = -11 }
```

On y voit des événements d'ordonnanceur et des appels systèmes. Des informations de contexte peuvent également être attachées à chaque événement, permettant ainsi de connaître le processus en cours d'exécution sur le processeur au moment où l'événement a été enregistré, les différents compteurs de performance sont également lus comme des informations de contexte, ce qui permet de voir l'évolution de chaque compteur au rythme de chaque événement.

Le traceur `SystemTap`

`Kprobes` fournit un point d'entrée pour exécuter du code arbitraire dans le noyau, utiliser cette technologie requiert une expertise en développement noyau, ce qui n'est pas forcément le cas de tous les utilisateurs voulant plus d'information sur leur système d'exploitation. C'est pourquoi le traceur `SystemTap` (Vara Prasad, 2005) a été conçu. Celui-ci vise la communauté des adminstrateurs système en leur fournissant un ensemble de scripts d'instrumentation qui s'interfacent automatiquement avec l'instrumentation statique fournie par `TRACE_EVENT`, mais également avec des points plus précis grâce à `kprobes`. Une limitation particulièrement gênante de `SystemTap`, dans un contexte où des traces volumineuses sont générées, est qu'il ne

possède pas de format de trace compact, les traces sont exportées sous une forme textuelle et impose des appels à une fonction équivalente à `printf` à chaque fois que des données doivent être exportées. De ce fait, ce traceur impose une charge sur le système et ne permet pas une analyse *a posteriori* détaillée. Par contre, les structures de données intégrées lui donnent la possibilité de calculer des statistiques personnalisées pendant la capture.

2.1.2 Traçage en espace utilisateur

Le traçage en espace utilisateur représente la prise d'information sur les applications avec les mêmes contraintes que la prise d'information au niveau noyau, c'est-à-dire de prendre le plus d'information sans interrompre le fonctionnement de l'application et avec le plus petit impact possible.

Dans ce domaine, on distingue deux catégories : les traceurs qui nécessitent une collaboration du noyau et les traceurs fonctionnant entièrement en espace utilisateur. Pour cette étude, nous nous intéressons uniquement aux traceurs qui permettent une approche où les traces noyaux et les traces en espace utilisateur peuvent être combinées et fonctionnant sous Linux.

`SystemTap` avec `utrace`

`SystemTap` peut être également utilisé pour tracer des applications en espace utilisateur grâce aux ajouts fournis par `utrace` (Corbet, 2007). `Utrace` est un ensemble de code non intégré au noyau, mais fourni par des distributions comme `Fedora` et `Red Hat`, qui remplace de manière transparente les fonctionnalités fournies par `ptrace`, tout en rajoutant une composante de traçage en espace utilisateur. Un point intéressant de ce traceur est que l'interface de programmation est standardisée et compatible avec les fichiers d'en-tête de `dtrace` (Bryan M. Cantrill et Adam H. Leventhal, 2004). Ce traceur nécessite cependant un appel système lors de chaque instruction tracée, ce qui implique un impact majeur dans les performances de l'application tracée.

`LTTng-UST`

Le traceur `LTTng`, avec le composant `User-Space Tracer` (Pierre-Marc Fournier, 2009), peut tracer des applications en espace utilisateur. Ce traceur fonctionne entièrement en espace utilisateur et les données sont exportées dans le format de trace de `LTTng`. Pour fonctionner il ne nécessite pas de modification particulière au noyau, mais il peut profiter de l'horloge de traçage de `LTTng` si elle est présente.

2.2 Traçage sur les autres systèmes d'exploitation

Le traçage n'est pas limité au système d'exploitation Linux. Bien que ce soit notre sujet principal, il est important d'étudier ce qui existe sur les autres systèmes d'exploitation.

2.2.1 Traçage sous Windows

Sous Windows, il existe également une infrastructure de traçage intéressante pour la résolution de problèmes. Le système `Event Tracing for Windows` est un environnement intégré pour le traçage au niveau applicatif et noyau. L'architecture est très similaire à celle de `LTTng` : on y retrouve des producteurs de traces qui envoient des événements stockés dans des tampons en mémoire. Ces événements sont ensuite acheminés aux clients souhaitant consulter les données en temps réel, et au besoin dans des fichiers de trace.

Dans le cas de la résolution de problèmes, les données produites par ce traceur sont très intéressantes et donnent rapidement un résumé des opérations les plus coûteuses en cours d'exécution. Par exemple, la figure 2.1 présente l'affichage de la bande passante sur le disque des fichiers les plus actifs.

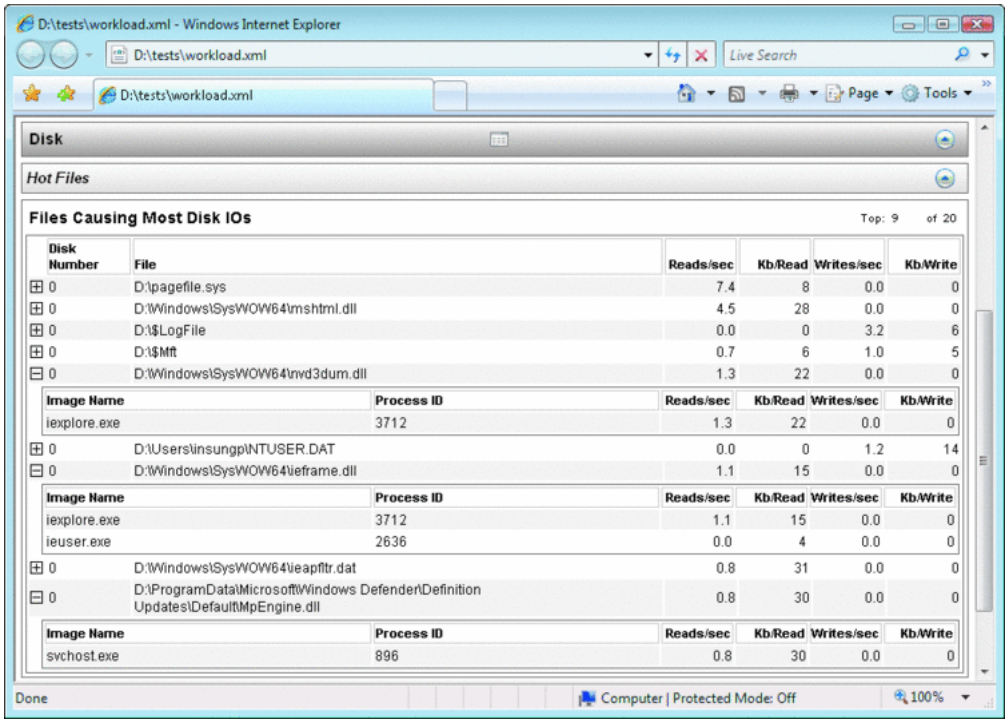

Figure 2.1 Bande passante par fichier dans ETW

Toutefois, comme il s'agit d'un système propriétaire, seule la documentation à haut niveau est disponible et il n'existe pas de documentation sur le fonctionnement interne du système.

2.2.2 Le traceur `DTrace`

Le traceur `DTrace` Bryan M. Cantrill et Adam H. Leventhal (2004) est un outil introduit sur Solaris 10 qui est conçu principalement pour la résolution de problèmes complexes. Depuis son apparition, il a été porté sur Mac OS X et FreeBSD. Ce traceur utilise des points d'instrumentation statiques et dynamiques dans l'espace noyau et utilisateur. Le langage «D» introduit avec `DTrace` est très intéressant car il donne rapidement aux développeurs des structures de données et des abstractions permettant de traiter les événements directement depuis des scripts en espace utilisateur (pas de compilation de module comme `SystemTap`) et sortir les résultats qui les intéressent. Toutefois, comme dans le cas de SystemTap, il est nécessaire de connaître le fonctionnement du code du noyau pour obtenir des données précises et pertinentes. Un autre problème est que `DTrace` n'a pas de format de trace binaire, il affiche les données déjà traitées dans un fichier ou à l'écran, mais il ne produit pas de fichier de trace compact contenant toute l'activité du système.

2.3 Virtualisation

La virtualisation, bien que popularisée depuis les dernières années par son utilisation croissante et l'infonuagique, existe depuis les balbutiements de l'informatique. La première mention du concept de virtualisation se retrouve dans un article de 1959 (Strachey, 1959), dans lequel l'auteur cherche à régler une solution pour monter un périphérique matériel sur une machine pendant qu'un programmeur travaille dessus. La technique qui découlera de cette recherche est maintenant appelée partage de temps (*time sharing*). Bien qu'il n'y ait pas vraiment de lien entre cette technique, qui s'apparente plutôt à un défi d'ordonnanceur, la virtualisation provient de ce besoin de maximiser l'utilisation d'une ressource physique et la partageant.

2.3.1 Virtualisation moderne

À l'heure actuelle, la virtualisation est devenue tellement essentielle et demandée, que les fabriquants de processeurs grand public ont développé des jeux d'instructions spécifiques pour augmenter la performance des machines virtuelles. Il s'agit de `Intel-VT` et `AMD-V`. Les performances de cette solution comparée à l'émulation sont étudiées en détail dans (Adams et Agesen, 2006). Dans cet article, les auteurs ont identifié grâce à des tests ciblés que, dans la plupart des cas, il existe un impact majeur sur le fait que les opérations de gestion de la mémoire sont interceptées par la machine hôte plutôt que d'être virtualisées et gérées nativement par la machine virtuelle. Cette remarque s'applique non seulement à la gestion de

la mémoire, mais également à la gestion des périphériques. À l'heure actuelle, seule une faible proportion du matériel externe tel que les cartes réseau ou les cartes graphiques possèdent des pilotes compatibles avec la virtualisation. De ce fait, les opérations d'accès à ces périphériques sont considérées comme privilégiées et requièrent que la machine hôte prennent en charge la communication.

Afin d'améliorer ce point, une approche courante est la para-virtualisation. Cette technique consiste à rendre une partie d'un système d'exploitation virtualisé coopératif avec la couche de virtualisation. De ce fait, ce composant fera des appels optimisés plutôt que d'essayer de faire des appels natifs et se faire intercepter par l'hyperviseur. Cette technique a été popularisée par l'hyperviseur Xen (Paul Barham, 2003) qui, le premier, a proposé une approche pour para-virtualiser le noyau Linux. Grâce à cette technologie, il est possible de faire fonctionner plusieurs machines virtuelles Linux sur un hôte Linux à très faible coût. Depuis, le support pour la virtualisation complète permettant de faire fonctionner des systèmes d'exploitation non-modifiés a été ajouté (Zhang et Dong, 2008).

Dans le cas de l'hyperviseur KVM (Avi Kivity, 2007), que nous allons étudier plus en détail au cours de ce mémoire, les pilotes de périphérique d'entrée/sortie les plus courants ont été para-virtualisés grâce à VirtIO (Russell, 2008). Cette approche permet de garder l'isolation permise par la virtualisation assistée par le matériel, d'exécuter nativement tout le code non-privilégié et, pour les opérations d'entrées et sorties intensives, demander la coopération du noyau de la machine physique en partageant des pages mémoires et des files de données entre les deux systèmes d'exploitation.

2.3.2 Contextualisation

Souvent associé à la virtualisation, la contextualisation est une méthode pour segmenter un système d'exploitation. L'objectif est d'isoler des ressources telles que le système de fichiers, la liste des processus et les connexions réseau, tout en partageant le même noyau, le même espace mémoire, les mêmes périphériques. De ce fait, les machines contextualisées profitent d'une performance maximale mais sont isolées les unes des autres. Cette technologie est très utilisée sous Solaris avec les `zones` (Daniel Price, 2004), sous BSD avec les `jails`, mais également sous Linux avec `Linux VServers`, `OpenVZ` et tout récemment `LXC`.

Contrairement aux technologies de contextualisation plus anciennes, `LXC` est entièrement basé sur du code entré dans le noyau officiel. Tout d'abord, le contrôle des ressources est basé sur les groupes de contrôle (`cgroups` Menage (2008)) qui permettent de limiter les ressources attribuées à des groupes de processus. Le pseudo système de fichiers `cgroups` est une organisation simple où il est possible d'associer des valeurs maximales aux ressources

telles que les périphériques de type bloc, les processeurs, la mémoire et le réseau. De plus, une liste explicite de périphériques autorisés ou refusés à un groupe se crée par la simple écriture dans les fichiers `devices.allow` et `devices.deny`. Pour créer un nouveau groupe, il est nécessaire de créer un nouveau répertoire dans le point de montage du système de fichiers `cgroups`, et ajouter dans le fichier `tasks` les identifiants des processus faisant partie de ce groupe.

L'isolation entre les différents groupes de processus est ensuite réalisée par la fonctionnalité des espaces de nom (`namespace` Biederman (2006)). Ce mécanisme permet à différents objets du système d'avoir le même nom dans des contextes différents. Les espaces de nom existants sont :

- UTS : le nom de la machine, la version du noyau, le nom de domaine ;
- Filesystem : la vision du système de fichier ;
- IPC : l'isolation du système global de communication inter-processus SYS V ;
- PID : la vision de la liste des processus en cours d'exécution ;
- User : la vision des utilisateurs existants et connectés sur le système ;
- Network : la vision des interfaces réseau, tables de routage et règles de pare-feu.

Lorsque combinés, les fonctionnalités d'espace de nom et de contrôle de groupe associées à une interface de contrôle, et des outils de gestions, sont la solution de contextualisation `LXC`. À l'heure actuelle, il existe encore des problèmes qui rendent l'utilisation de cette technologie risquée dans le cadre de serveurs de production, mais les problèmes sont connus et en train d'être résolus. Spécifiquement : les entrées globales du système de fichier `proc` ne sont pas filtrées dans les contextes. Il est donc possible depuis n'importe quel contexte de consulter ou modifier des paramètres communs à la machine. De plus, les appels systèmes ne sont pas filtrés, donc une machine contextualisée peut impacter fortement l'hôte et les autres contextes si elle choisit par exemple de charger un module noyau. La solution à ce problème est en cours d'intégration sous la forme de `seccomp` Corbet (2009).

`LXC` est la technologie retenue et acceptée par la communauté du noyau Linux pour la contextualisation, ses composants de base sont intégrés au noyau et les outils de contrôle sont des interfaces aux espaces de noms et aux groupes de contrôle.

Contrairement aux technologies de virtualisation complètes, la contextualisation apporte donc un moyen à faible coût d'isoler des groupes de processus tout en partageant le même noyau et les mêmes ressources physiques.

Un aspect particulièrement intéressant de contextualisation pour les technologies embarquées est le fait qu'il s'agit du seul moyen efficace à l'heure actuelle pour isoler des ressources et de compartimenter les applications et informations. Les instructions de virtualisation comme celles introduites par Intel et AMD ne sont pas disponibles sur l'architecture

ARM, donc la technologie de contextualisation va devenir de plus en plus populaire sur les plateformes embarquées telles que les téléphones intelligents.

2.4 Traçage de la virtualisation

Un des objectifs de ce mémoire est d'étudier le traçage des solutions de virtualisation, tant au niveau du système d'exploitation hôte, que du système d'exploitation virtuel, que des couches d'interfaces et des applications en espace utilisateur dans les deux mondes.

Dans l'article Heidari *et al.* (2008), le problème du traçage dans une machine virtuelle est abordé. Dans cette étude, les auteurs ont modifié l'hyperviseur Xen, fonctionnant en para-virtualisation, pour générer des événements sous certaines conditions. L'étude consiste principalement à mesurer l'impact de cette solution ainsi que l'impact de la para-virtualisation en comparant le résultat de tests de performance avec les résultats natifs. Pour l'instrumentation, la méthode retenue est celle d'utiliser des `hypercalls`, qui sont des appels privilégiés exécutés depuis une machine virtuelle pour communiquer avec l'hôte, pour extraire les données et par la même occasion les synchroniser avec ses traces. Cette méthode, bien que fonctionnelle et permettant une synchronisation des traces sur plusieurs niveaux, impose une charge non-négligeable dans les deux environnements et n'est pas applicable en production.

La sécurité est également un domaine très intéressé par la virtualisation, principalement pour l'isolation qui est apportée et la facilité de revenir à un état connu. Un autre aspect intéressant de la virtualisation pour la sécurité est le fait que toutes les opérations, même de très bas niveau, peuvent être interceptées par le système d'exploitation hôte. Ainsi, il est possible d'avoir une trace exacte de l'exécution d'un programme sans même qu'il ne puisse le détecter. C'est exactement ce principe qui a été suivi avec le développement de ReVirt (George W. Dunlap, 2002). Dans cette étude, l'objectif est d'identifier les logiciels malveillants et les cas d'intrusion en enregistrant suffisamment de données sur l'exécution du système d'exploitation. La capture des traces est faite à haut niveau : les auteurs utilisent le démon d'écriture de journaux (`syslog`) pour écrire des informations qu'ils jugent pertinentes. L'intérêt de cette recherche est qu'ils écrivent les traces dans des données qui sont exportées sur l'hôte et non enregistrées dans la machine virtuelle. Toutefois, les interruptions, les opérations de l'ordonnanceur et la plupart des autres actions du noyau sont ignorées car jugées inutiles dans le cas des analyses qu'ils souhaitent réaliser.

Dans l'article Zeng et Hao (2009), les auteurs utilisent le traçage de la virtualisation pour identifier des problèmes de latence dans la couche réseau. Encore une fois, on constate que le traçage est utile pour les développeurs. Toutefois, la méthodologie utilisée est différente,

car dans ce cas ils utilisent le traceur `SystemTap` afin de savoir quand le code de la pile réseau entrante est appelé, et ensuite utilisent le dévermineur `GDB` pour arrêter l'exécution et consulter les valeurs de paramètres passés aux fonctions. Cette méthodologie est différente de celle que nous cherchons à mettre en place car elle ne s'applique pas en production, elle vise les développeurs de la couche réseau de KVM, quand ils travaillent sur des machines de développement.

2.5 Résolution de problèmes

2.5.1 Méthode traditionnelle

La résolution de problème se caractérise par la recherche de solutions lorsqu'un composant ne fonctionne pas comme prévu. Pour ce faire, la première étape est d'avoir assez d'informations pour être en mesure de faire un jugement avisé. De nombreux outils existent pour résoudre des problèmes et la plupart du temps ils permettent d'obtenir les résultats attendus. Il y a plusieurs catégories d'outils permettant d'obtenir des informations sur un système et ainsi aider à la résolution de problèmes :

- Statistique : ces outils se servent des compteurs sur le système pour produire des moyennes et des vitesses (par exemple `top`, `vmstat`) ;
- Tendance : ces outils cumulent les compteurs et les affichent en fonction du temps (par exemple `Cacti`, `Munin`, `Collectd`) ;
- Données : ces outils permettent de rentrer dans le contenu des données échangées entre différents sous-systèmes (par exemple `tcpdump`).

Ces outils, essentiels à la recherche et bien connu des administrateurs, sont souvent suffisants pour les problèmes communs. Toutefois, pour certaines classes de problèmes, ils sont inefficaces voire même nuisibles. Pour ces cas, où des données précises doivent être collectées et pendant une durée indéterminée, le traçage semble être une bonne solution pour combler les lacunes présentes dans les outils à plus haut niveau.

2.5.2 Le traçage pour la résolution de problèmes

Un phénomène observé, dans la recherche de solutions pour automatiser la découverte de problèmes dans les systèmes d'exploitation, est le manque de solutions génériques et en même temps détaillées. Il y a dans la littérature de nombreux articles visant à régler des problèmes précis, mais très peu pour identifier des pistes de recherche et fournir des résultats détaillés lorsque nécessaire.

L'article Ruan et Pai (2004) est intéressant car il cherche à apporter lors du retour de chaque appel système des informations de performance. Cette solution permet à l'application de s'adapter et éventuellement créer des alertes sur certaines conditions détaillées mais, avant tout, elle permet de mesurer précisément pour chaque appel système l'impact et le délai causé. Toutefois, cette solution implique une coopération des applications et ne permet pas d'identifier les problèmes liés à l'interaction entre plusieurs processus. Dans le contexte de la recherche de problèmes à haut niveau, cette méthode ne s'applique pas.

Le domaine du calcul en grappe et à haute performance est aussi à la recherche des facteurs ralentissant l'exécution de certaines tâches, c'est pourquoi dans l'article Sushant Sharma et Maccabe (2005), les auteurs ont également choisi d'extraire des informations de performance des appels systèmes. Dans cette recherche, ils adaptent le traceur `Linux Trace Toolkit` pour collecter plus d'informations sur les appels systèmes et ainsi fournir des données précises sur le délai survenu pendant l'exécution de l'appel système. Dans ce cas, l'approche utilise du traçage, reste générique et peut donc s'appliquer pour une grande variété d'applications. Toutefois, les données fournies sont encore une fois trop détaillées pour orienter un administrateur à trouver des pistes.

2.6 Conclusion de la revue de littérature

Le domaine de la virtualisation grand public et du traçage étant des domaines relativement nouveaux, on constate que peu de recherche a été faite sur l'intégration des deux sujets. Les traceurs ont beaucoup évolué jusqu'à devenir des composants intégrés au noyau Linux, la virtualisation aussi, mais les rares cas où les deux domaines se croisent, il s'agit d'usage *ad hoc* conçu pour résoudre des problèmes spécifiques par les développeurs des solutions.

Concernant l'identification de problèmes sous Linux, on voit à travers les recherches déjà effectuées que la plupart des solutions proposées se concentrent sur la précision et le détail des informations fournies. Avoir des données aussi précises est essentiel pour extraire des informations exactes, mais il est nécessaire pour cela de connaître, au moins approximativement, l'origine du problème. À l'opposé, dans l'industrie, avec les logiciels intégrés et connus des administrateurs système, on extrait facilement des données à haut niveau qui donnent de l'information générique sur une période donnée. Il existe donc un besoin d'avoir des outils et des méthodologies efficaces pour combler ce manque entre les données génériques et les informations détaillées.

Chapitre 3

MÉTRIQUES DE PERFORMANCE SYSTÈME

La recherche et la résolution de problèmes complexes passe tout d'abord par la mesure précise de facteurs permettant d'identifier des comportements suspects. C'est pourquoi dans cette section nous allons étudier en détail les métriques offertes par le traçage pouvant aider la résolution de problèmes. Tout d'abord, nous étudierons les statistiques disponibles dans LTTV, puis nous détaillerons les métriques additionnelles identifiées dans des environnements réels avec la collaboration de l'entreprise Révolution Linux.

3.1 Statistiques de `LTTV`

Dans `LTTV`, l'analyseur graphique de traces de `LTTng`, un module est responsable du calcul des statistiques. Ces statistiques concernent la trace complète et sont disponibles de manière globale (quantité d'événements), par mode, par processeur et par processus.

3.1.1 Statistiques globales

Les statistiques disponibles de manière globale sont :

- *events count* : le nombre total d'événements dans la trace ;
- *cpu time* : le total du temps d'exécution de chaque processeur ;
- *elapsed time* : le total du temps d'exécution de chaque processus ;
- *cumulative cpu time* : le temps total d'activité des processus (incluant le temps où les processus étaient en espace noyau).

Le compteur d'événements

Il s'agit du nombre total d'événements de tous types. Nous allons le voir ci-dessous, il peut être global à la trace, ou groupé par mode, processus et processeur.

Temps CPU

Il s'agit du cumul du temps de calcul total sur chaque processeur. Cette valeur est calculée par une addition du temps où chaque processeur était actif (tous les états sauf *idle*) pendant la période voulue (temps de la trace ou la durée de vie d'un processus). Ainsi, la limite supérieure de ce calcul est le produit du nombre de processeur avec la durée.

Temps écoulé

Pour un processus, cette valeur est le temps réel d'exécution. Donc ce temps comprend le temps passé en espace utilisateur, le temps passé dans le noyau et le temps d'attente. Donc il s'agit de la durée de vie d'un processus.

Temps cumulé

Pour un processus, le temps cumulé est la somme du temps passé en espace utilisateur et en espace noyau. La différence avec le temps écoulé, c'est que ce compteur ne prend pas en considération le temps d'attente, ce qui permet de donner une valeur très précise sur le temps réel où le processus a été actif.

3.1.2 Statistiques par mode

Les modes de `LTTV` sont des états dans lesquels un processus ou un processeur peuvent se retrouver en fonction des événements survenus. Ils sont calculés lors de l'analyse. Les modes disponibles sont :

- IRQ : lorsqu'une interruption est reçue par un processeur, le processus en cours d'exécution passe dans l'état IRQ, pendant le temps que le noyau traite l'interruption.
- SYSCALL : lorsqu'un processus exécute un appel système, il passe dans l'état SYSCALL pendant le temps que le noyau traite l'appel système.
- SOFTIRQ : lorsqu'une interruption est survenue, le gestionnaire d'interruption exécute habituellement un traitement rapide pour transfèrer les données de l'interruption à un gestionnaire moins prioritaire afin d'être prêt le plus rapidement possible à traiter les autres interruptions pouvant survenir.
- USER_MODE : lorsque le processus courant exécute du code en espace utilisateur.
- MODE_UNKNOWN : lorsqu'il est impossible d'après les événements reçus de connaître le mode d'un processus.

Un processus est toujours dans un de ces modes, mais le mécanisme de mode fonctionne à l'image du système d'exploitation : sous forme de pile. En effet, lorsqu'un processus fonc-

tionne en espace utilisateur et exécute un appel système (événement `syscall_entry`), il empile l'état SYSCALL à son état courant. De ce fait, lorsque l'appel système est terminé, l'événement `syscall_exit` le ramène au dernier état sur la pile : USER_MODE.

Les statistiques disponibles dans `LTTV` sont uniquement accumulées dans le mode courant : le mode en haut de la pile.

La figure 3.1.2 présente un exemple de statistiques groupé par modes. On y voit les événements qui ont amené le système en mode SYSCALL ainsi que les événements qui sont survenus pendant qu'un processus ou un processeur étaient dans ce mode.

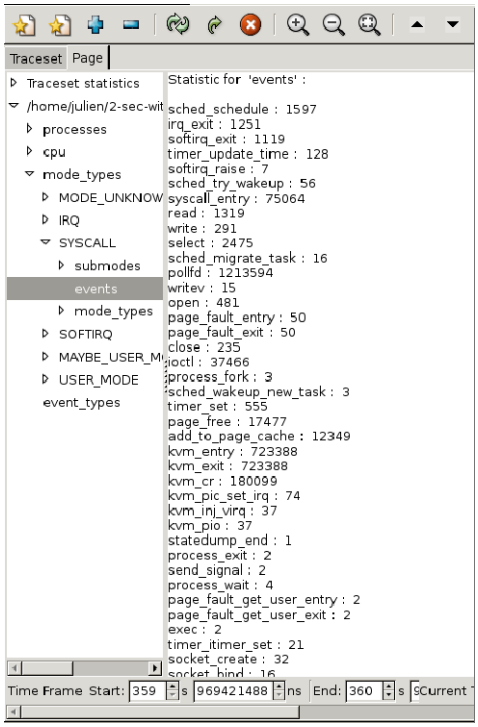

Figure 3.1 Statistiques de LTTV groupées par Mode

3.1.3 Statistiques par processeur

Les statistiques par processeur regroupent les événements en fonction du processeur sur lequel ils se sont exécutés. Ce regroupement est fait de manière globale mais également par mode. Donc il est possible de savoir combien d'événements se sont exécutés sur un processeur spécifique dans un mode spécifique. Cette information est particulièrement intéressante sur les systèmes temps réels où les interruptions sont traitées par des processeurs spécifiques. Avec cette information, il est possible de vérifier rapidement si les paramètres sont respectés.

3.1.4 Statistiques par processus

Les statistiques par processus permettent de consulter les statistiques génériques par processus (les différents temps CPU limités à l'exécution du processus choisi), grouper les événements par type et d'accéder au compteur de type d'événement par mode et par processeur. Encore une fois, ces statistiques sont utiles pour valider le temps passé par processus dans chaque mode, mais également pour identifier rapidement les migrations de processeur qui peuvent être des causes de ralentissement dans le temps d'exécution. L'image 3.1.4 présente un exemple de l'affichage des statistiques par processus.

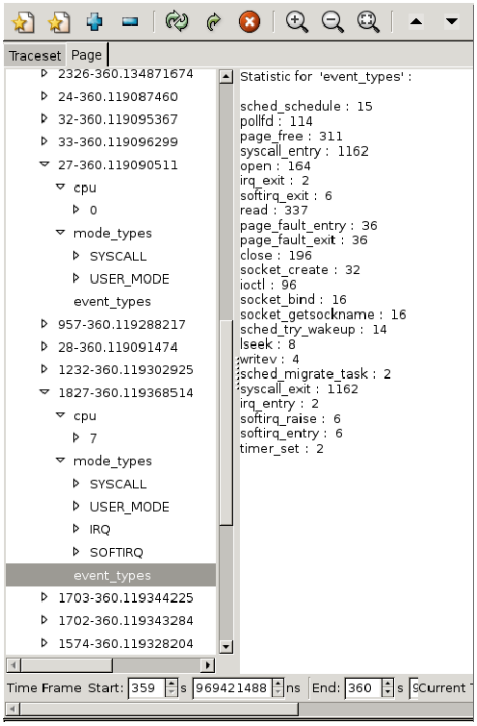

Figure 3.2 Statistiques de LTTV groupées par processus

3.1.5 Conclusion sur les statistiques de LTTV

Les statistiques produites par LTTV donnent rapidement une vue d'ensemble du temps d'exécution détaillé de chaque processus, de tous les états dans lesquels un processus ou un processeur ont été actifs, et de tous les processeurs ayant servi à l'exécution d'un processus. Ces statistiques donnent des informations très pertinentes sur l'activité de la machine. Toutefois, il s'agit de données globales basées sur la trace complète, donc il peut être assez difficile de déterminer un moment précis où une action suspecte est apparue, particulièrement dans le cas d'une longue trace.

Dans la section suivante, nous allons détailler les métriques que nous souhaitons obtenir.

3.2 Automatisation de la recherche de problèmes

Étant donné que la recherche dans ce projet s'est effectuée en partenariat avec l'entreprise Révolution Linux, qui a des besoins très concrets en terme de recherche de problèmes complexes sur des serveurs en production, une liste de métriques pouvant être obtenues par le traçage a été élaborée. Les éléments dans cette liste sont des besoins en terme d'analyse automatisée qui vont améliorer l'efficacité du travail de résolution de problème.

Cette liste est présentée ici car il s'agit d'un travail de réflexion qui est arrivé jusqu'à l'élaboration d'algorithmes, et dans certains cas d'implémentation dans `LTTngTop`. Les éléments non implémentés de cette liste font partie des améliorations prévues au logiciel. Dans certain cas, des sondes pour `LTTng` devront être écrites, mais toute l'instrumentation côté noyau a été vérifiée et est disponible sur un noyau standard à partir de la version 2.6.35.

3.2.1 Valider les contraintes d'affinité

la migration d'un processus vers un autre processeur impose une charge sur le système. Dans le cas de la virtualisation, associer un processeur virtuel à un processeur physique est une pratique courante afin d'éviter les comportements comme ceux observés sur la figure 4.2. Il s'agit également d'une pratique courante dans les environnements temps réels où les processus critiques et les interruptions sont séparés afin d'avoir une meilleure garantie du temps d'exécution. La validation se fait en entrant les contraintes à l'avance et en vérifiant les événements de migration de processus `sched_migrate_task`.

3.2.2 Identifier les changements de fréquence

Identifier l'application ou le sous-système qui crée le plus de changement d'état au niveau de la gestion d'énergie (comme fait par l'application `powertop`), peut être intéressant lorsque l'on consulte des traces *a posteriori*. Avec cette analyse, il est possible de connaître rapidement le processsus causant le plus de changement de fréquence du processeur et ainsi, sans nécessiter un accès à la machine, déterminer des pistes de solutions pour augmenter la durée de vie de la batterie. Cette analyse est possible grâce aux événements de la classe `power`.

3.2.3 Identifier la répartition des distances de recherche (*seek*)

Une des causes principales de lenteur sur un serveur est liée aux accès au disque dur, les opérations de recherche sur le disque impliquent un déplacement physique de la tête de lecture : plus ces recherches sont proches (en terme d'adresse de secteur à lire), plus la distance est courte et donc moins l'impact se fait sentir sur le système d'exploitation. L'objectif de cette métrique est d'identifier les recherches qui impliquent des déplacements majeurs et ainsi identifier les processus qui se nuisent mutuellement en demandant au même instant des lectures à des secteurs fortement éloignés. Cette analyse est possible grâce aux événements de la classe `block`.

La représentation produite par le script `SystemTap` de l'annexe A visible sur la figure 3.3 est très intéressante car elle permet de visualiser rapidement la répartition des distances de recherche sur le disque. Toutefois, elle ne permet pas de donner plus d'information concernant les processus responsables de ces requêtes.

3.2.4 Calculer la latence d'un composant d'entrée/sortie

Les événements de bas niveau nous permettent d'obtenir le délai exact entre l'envoi d'une demande et la réception de la réponse, avec cette mesure nous pouvons ainsi déterminer quand une ressource fonctionne dans des normes acceptables et quand elle dépasse certaines limites configurables. La latence est bien souvent ce qui caractérise le phénomène de lenteur d'un système : le délai de réaction du système. Avec un calcul fiable de la latence sur des composants comme le disque dur, il est possible d'avoir des données très précises sur le temps moyen d'attente sur ce composant ainsi que les processus impactés par ce délai. Encore une fois, cette analyse est possible grâce aux événements de la classe `block`.

3.2.5 Présenter la fréquence des événements

Que ce soit à l'échelle globale du système, ou limité à un processus en particulier, avoir une répartition du nombre d'événements permet souvent d'identifier une piste à explorer pour trouver la source d'un problème. Dans ce cas, l'analyse se base sur les événements activés. Une bonne piste de départ est les appels systèmes. Souvent, `strace` donne rapidement la solution à un problème car l'utilisateur peut identifier facilement les appels systèmes exécutés le plus souvent. Avoir cette information avec la trace de manière concise est un atout majeur.

```
Device: dm-1
     value |-------------------------------------------------- count
-134217728 |                                                    0
 -67108864 |                                                    0
 -33554432 |@                                                   1
 -16777216 |                                                    0
  -8388608 |                                                    0
  -4194304 |                                                    0
           ~
         2 |                                                    0
         4 |                                                    0
         8 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@         42
        16 |@@                                                  2
        32 |                                                    0
        64 |@                                                   1
       128 |                                                    0
       256 |                                                    0
       512 |                                                    0
      1024 |                                                    0
      2048 |@                                                   1
      4096 |                                                    0
      8192 |                                                    0
     16384 |@                                                   1
     32768 |@                                                   1
     65536 |                                                    0
    131072 |                                                    0
           ~
  33554432 |@@                                                  2
  67108864 |                                                    0
 134217728 |                                                    0
```

Figure 3.3 Représentation des distances de recherche sur le disque par SystemTap

3.2.6 Présenter l'utilisation des ressources physiques

Avec les informations obtenues par le traçage, il est possible de présenter une répartition de l'utilisation des ressources physiques et les trier par processus, utilisateur, machine virtuelle, descripteur de fichier. Grâce à cette analyse, il est possible de représenter rapidement à l'utilisateur l'état de chaque ressource et les principaux consommateurs. La bande passante disque ou réseau liée à un descripteur de fichier est une information très intéressante. En plus d'apporter des pistes de résolution de problèmes, cette analyse permet également de prévoir l'impact qu'un nouveau composant aura sur le système et les conflits pouvant survenir dans les cas de mise en production. L'évaluation du pourcentage d'utilisation d'une ressource provient de l'analyse de latence.

3.2.7 Analyser les verrous et problèmes de contention

La gestion des verrous dans les applications est complexe à valider car ajouter des informations peut changer le comportement de l'application. Cette analyse sert à identifier avec précision les périodes de temps où un processus est bloqué à cause d'un autre processus possédant le verrou, et le temps de ce blocage.

3.2.8 Conclusion

Les idées et métriques présentées jusqu'ici sont des moyens d'automatiser la résolution de problèmes sous Linux. Les nouveaux points de mesures extraits des informations de traçage, peuvent être intégrés avec les outils de supervision, et ainsi en augmenter l'efficacité et le temps de réponse en cas de problème sur un système.

Chapitre 4

EFFICIENTLY TRACING ACROSS THE HYPERVISOR LAYERS

Authors

Julien Desfossez
École Polytechnique de Montréal
julien.desfossez@polymtl.ca

Michel R. Dagenais
École Polytechnique de Montréal
michel.dagenais@polymtl.ca

Submitted to

Operating System Review

Keywords

Tracing, Virtualisation

4.1 Abstract

Virtualization is now a mandatory feature for any data center and server deployment. We use more and more virtual machines (VM) because of the flexibility they provide and the always decreasing impact on performance. In modern architecture, a hypervisor (or Virtual Machine Monitor) is a component running inside the host kernel and responsible for attributing physical resources to the VMs. Usually it is coupled with a user-space component that manages the user interface to control and interact with the VMs (QEMU in the case of KVM). To thoroughly analyse the behaviour of a running VM with respect to its interactions

with the host kernel, we need a tracer that can record and analyse the traces at kernel and user-space levels. Once we have such traces, we need a proper methodology to analyse this information. This paper introduces a new CPU state to represent the concept of virtual CPU (VCPU) providing a way to analyse the interactions between a VM and its environnement. Moreover, we introduce and evaluate a new clock source for the Linux kernel, designed to assist a pure user-space tracer such as LTTng-UST to record traces with accurate timestamp, and then merge at analysis time with the kernel traces, avoiding costly system calls and synchronisation methods.

4.2 Introduction

Virtual machines are now at the core of most new server deployments, and an efficient collaboration with the host and guest resources is essential in order to achieve good performance. Since the advent of hardware assisted virtualization Adams et Agesen (2006), VMs are attaining performances close to the hosts, and now we clearly see that a limiting factor is this constant interaction between the VMs and the host kernel. Whether it is for accessing hardware, privilege issues or just scheduling, these kinds of heavy context-switches have a considerable impact on the VMs speed and latency.

In a production environment, or during development, when we try to diagnose a VM performance issue, we need an accurate method to analyse its behaviour with a minimum impact on the host and guest operating systems. Tracing is the solution for gathering a maximum of information without stopping or breaking into the traced processes. For example, we want to know everytime a virtual CPU exits from a VM (the `vm_exit` instruction) and the reason of this exit. Using this information combined with the other traces of the operating system, we have a way to determine if this exit is normal or could be avoided by tweaking some parameters. For the KVM hypervisor, all of this is already possible with `perf` Edge (2009b) and `ftrace` Edge (2009a) thanks to the `TRACE_EVENTS` defined inside the kernel Rostedt (2010). But these events happen really frequently and what is missing is a comprehensive and practical methodology to analyse traces generated from the hypervisor, the other operating system components and the user-space control tools.

In this paper, we introduce in section 5.2 previous work related to this topic, in section 4.4 a new methodology and abstraction to analyse hypervisor traces, in section 4.5 a new Linux clock source to provide a way for user-space tracer to record traces that can be merged accurately with kernel-space traces, without requiring any interaction between the two layers at run-time, then in section 5.5 we present the benchmarks of this trace clock and the result of a complete multi-layer analysis, and finally in section 5.6 we discuss the next steps for

this research and conclude in section 5.7.

4.3 Related Work

In Heidari *et al.* (2008), Heidari et al. extended the Xen hypervisor to support tracing across layers. In order to do so, they overrided the Xentrace trace sites with LTTng markers and benchmarked the impact of this tracing. This solution provides an interesting way to extract information from the hypervisor back to the host kernel but it works thanks to `hypercalls`, which is costly and generates additional `vm_exit` operations.

In Zeng et Hao (2009) Zeng et al. propose a way to trace and debug latency residing inside the ingress network stack. This methodology is designed for KVM developpers to find bottlenecks. It is based on the SystemTap tracer and GDB to help find precisely where in the code is the latency. Although it is working, this solution has not the same focus as ours : it is not designed for production environment and introduces a very important overhead.

4.4 Tracing the hypervisor

4.4.1 KVM hypervisor instrumentation

KVM is a full virtualization solution that has been in the mainline Linux kernel since 2.6.20. Full virtualization means that it can, on supported hardware architecture, run unmodified guest operating systems. It is composed of two kernel modules : the first is generic and responsible for the core virtualization infrastructure, and the other is specific to Intel or AMD processors.

The `TRACE_EVENT` macro is an easy way for kernel developpers to instrument statically their code in order to record and retrieve data at specific sites for later analysis. As of Linux kernel 2.6.37 the KVM code is instrumented at 21 strategic locations and tools like `perf` and `ftrace` use these events to give KVM developpers a detailled report about what is happening when a VM is running.

For example, when recording one minute of statistics when a VM is booting up, `perf` produces an output similar to this one (sorted manually and most counters with a value of zero have been removed for clarity) :

```
$ sudo perf stat -e 'kvm:*' -a sleep 1m
Performance counter stats for 'sleep 1m':
```

```
1,284,583 kvm:kvm_entry
1,284,521 kvm:kvm_exit
921,990   kvm:kvm_page_fault
319,506   kvm:kvm_inj_exception
146,792   kvm:kvm_emulate_insn
91,372    kvm:kvm_userspace_exit
86,201    kvm:kvm_pio
66,843    kvm:kvm_cr
42,034    kvm:kvm_cpuid
30,104    kvm:kvm_mmio
21,179    kvm:kvm_apic
17,536    kvm:kvm_fpu
9,505     kvm:kvm_apic_accept_irq
9,125     kvm:kvm_inj_virq
6,718     kvm:kvm_set_irq
3,904     kvm:kvm_pic_set_irq
3,904     kvm:kvm_ioapic_set_irq
2,814     kvm:kvm_msi_set_irq
2,513     kvm:kvm_apic_ipi
1,016     kvm:kvm_ack_irq
225       kvm:kvm_msr
4         kvm:kvm_age_page
0         kvm:kvm_hypercall
0         kvm:kvm_hv_hypercall
```

In this example, we only record statistics about the KVM events, but we can also generate statistics for the whole operating system. One detail to take into account is that these statistics are created from the trace events in the KVM kernel code, which means that if we run multiple VMs on the same physical host, we have no way to isolate these statistics for a particular instance. Moreover, judging from the amount of KVM entry and exit events which should be equalt, we know that the tracer lost some of events generated.

Sometimes, to understand the behaviour of a VM, it is more interesting to read the timeline of the events happening in KVM and also on the whole host system. The `ftrace` kernel tracer generates a timeline and displays precisely what is happening on the system over time. In the next example, we activate all trace events available on our machine and present a small window of the output :

```
kvm [000] 3031.712637: kvm_entry: vcpu 0
cat [001] 3031.712638: sys_exit: NR 0 = 4086
cat [001] 3031.712638: sys_enter: NR 1
          (1, 152d000, ff6, 0, 7f3429e01e60,
          7f3429e01eb0)
kvm [000] 3031.712639: kvm_exit:
          reason EXCEPTION_NMI
          rip 0xffffffffa0025cb1
          info ffffc9000031403e
          80000b0e
cat [001] 3031.712639: ext4_da_write_begin:
          dev 8,2 ino 1990 pos 32439278
          len 1042 flags 0
kvm [000] 3031.712639: kvm_page_fault:
          address ffffc9000031403e
          error_code b
cat [001] 3031.712639: kmem_cache_alloc:
          call_site=ffffffffa010e7bb
          ptr=ffff88010ff72330
          bytes_req=24 bytes_alloc=24
          gfp_flags=GFP_NOFS
cat [001] 3031.712640: kfree:
          call_site=ffffffffa010e638
          ptr= (null)
```

We can see in that example that the processes `cat` and `kvm` are running in parallel on two separate processors. For KVM debugging, the information is also particularly relevant because we see everytime the VM is actually running, everytime it exits to the host and the reason of this exit.

These are the two tracers integrated in the mainline Linux kernel. We have seen that they give quickly a good overview of what is happening on the system and also provide a way to dig deeper when needed.

However these tracers are not integrated together and still they are complementary ; when an administrator or a developper needs to analyse a performance problem with a VM, he first needs access to detailled statictics such as the one provided by `perf` and then, when he gets an idea of what is happening, he may want to look at the timeline with `ftrace` and identify the interactions between the VM and the host to see what could be optimized.

When we look at the timeline generated by `ftrace`, we see that there are many events : on an idle dual-core system running an idle Linux VM, we receive more than 2.7 million events in 10 seconds with all trace events active. This means that to identify the cause of a problem, we have to know precisely when it happened. The graphical interface `KernelShark` is a tool designed to help sort through the data generated by `ftrace`. It provides a graphical timeline of the activity of each process on each CPU, it also allows the user to filter on particular processes and events and zoom into the trace. Thanks to this tool, we can identify when the KVM process is active but still we don't know in which state it is, for example it could be running in user-space to handle the user interactions, in kernel space emulating privileged instructions, or running the actual VM code natively. Moreover if the problem we are investigating is caused by a hardware latency, we may spend a lot of time finding it, since these kinds of problems are linked to asynchronous events, and manual correlation is difficult. For example, when we request data from the hard drive, the kernel blocks our current task (if we are doing synchronous access) but continues to execute others while the data is being retrieved.

`LTTng` M. Desnoyers (2006) is an out-of-tree kernel tracer which also uses probes connected to the `TRACE_EVENTS` already defined inside the mainline kernel code. The events recorded are the same as the one we observe with `ftrace`, provided the custom probes are created. This tracer comes with a graphical traces analysis tool (`LTTV`) which gives quickly statistics about the events recorded, a control flow viewer and a resource viewer. The control flow viewer allows the user to see the state of each process ; in this view, a process can be in one of the following modes : system call, trap, irq, softirq, waiting for I/O or CPU, user-space, zombie, waiting for fork, exit, dead, and finally unknown (in case the state cannot be determined with certainty). In the resource viewer, the processors, block devices and IRQs are represented following the same timeline, which gives an overview of the resources usage. Thanks to this combination of features, we have a way to identify the cause of a blocked process and the interactions between processes and ressources.

Judging from this analysis, we see that `LTTng` is the tracer more suited to allow an efficient analysis of KVM behaviour, because it provides statistics about a trace and has the tools to properly analyse the chain of events and the impact on the physical resources. To make the analysis more efficient though, we would like to know more about the KVM events and especially have a way to graphically represent when a processor is executing code natively for a virtual machine. That way we could gather statistics about the exact amount of time a VM is really executing its own code (not just the KVM process).

4.4.2 Introducing the VIRT CPU state

The state system in LTTV allows a process or a physical resource to be represented differently depending on its state. The state of a process or a resource is calculated during analysis by looking at the events that occurred. For example, following a `syscall_entry` event, a process is considered in `SYSCALL` mode until a `syscall_exit` occurs. This state system is particularly useful to produce statistics based on the amount of time a process or a resource spent in a particular mode, and also to display a different colour based on this state. That way the user can see quickly the transitions that occurred while a process was running.

For virtualization, the already defined states are not enough to get a clear overview of a VM behaviour. When running hardware-assisted virtual machines, a particular set of instructions (VMX and HVM for Intel and AMD respectively) allows a processor to be dedicated to a VM for non-privileged instructions. In the case of KVM, each `kvm` thread represents a virtual CPU (VCPU) of a guest. Every time one of this thread is scheduled in, it executes code of the VM. The hypervisor is in charge of loading the environnement of the VM as seen on last exit and executing the `vmrun` instruction. As long as the VM executes non-privileged code, and is not preempted or voluntarily yields the processor (instruction `HLT`), it executes natively its code on the physical processor. During that time, we know that the VM is really running its own code. But there is a considerable period of time where a `kvm` thread is scheduled in but is only managing the VM and not running it. For example, when dealing with interrupts and various user interactions, a `kvm` process is active but the VM is not actually running.

The new LTTV state `VIRT` allows to represent this state where a KVM process is actually executing code inside a virtual machine. Thanks to this new state, we provide the user access to more accurate time of execution of a VM, and a graphical view of the processor entry and exit inside the VM. Moreover using the built-in filtering mechanism of LTTV, it is possible to filter on specific processes and limit these features to specific VM instances (when multiple VMs are running) or even specific VM virtual CPU.

Since LTTng kernel tracer is based on the `TRACE_EVENTS` defined in the kernel, the first step to implement this new state was to connect probes to the KVM events to get an output similar to this one in LTTV text-dump mode :

```
kvm.kvm_entry: { vcpu = 0 }
kvm.kvm_exit: { reason = 48
    [ept_violation] }
kvm.kvm_page_fault: { address = feffc000,
    error_code = 81 }
```

Once the `kvm_entry` and `kvm_exit` events got recorded in the LTTng traces, we integrated these events in the `LTTV` state system. The result of this integration is detailed in section 5.5.

4.5 Tracing across layers

Operating system privileges are often represented as rings : ring 3 is the user-space layer, ring 0 is the kernel (or privileged) layer, rings 1 and 2 are reserved for device drivers but are not actually implemented in Linux. With the apparition of the hypervisor, since a VM can execute natively its own ring 0 code without impacting the host or other VMs, a new ring conceptually below ring 0 appeared : ring -1.

To record a complete trace of a virtual machine, we have to trace at various levels of execution :

- The host's user-space (QEMU)
- The host's kernel module
- The hypervisor instructions
- The eventual paravirtualized drivers
- The guest kernel
- The guest user-space

In Linux, the kernel, the drivers and the hypervisor work in the same address space and they can communicate directly with each other so they are physically on the same layer. The communication between the user-space and the kernel-space goes through system calls and vDSO. The communication between a VM and the host is done through hypercalls and synchronized access to shared memory pages and registers. These communications between layers are costly and even though it is a good way to ensure that multi-level traces are synchronized, a tracer should not add such an overhead to a production system.

An alternative approach, in order to synchronise multi-level traces, is to record them independently, ensure we have a reliable shared clock source and merge the traces at analysis time.

In the following sections we will describe the LTTng trace clock and how we modified it to allow user and kernel-space traces to be synchronised efficiently.

4.5.1 LTTng trace clock

Various clock sources are available in Linux ; all of them have their pros and cons. On the x86 architecture, there are various hardware or emulated devices available for timekeeping

(PIT, RTC, HPET) Amsden (2010), but the Time Stamp Counter (TSC) provides the highest granularity. The TSC mode of operation is the simplest : it counts instruction cycles issued by the processor, it can be accessed from kernel and user-space with the `RDMSR`, `RDTSC` or `RDTSCP` instructions. However since it is part of the processor and not a centralized external device, there are many synchronisation issues possible with multi-processor systems. Using this clock as a system wide clock source comes with a lot of challenges and considerations Amsden (2010) but, in the case of tracing, the problem is different : the granularity and monotonicity are the most important criterion because we need to make sure the events we record are timestamped at the highest granularity possible so they can be serialized at analysis time. The TSC is the most appropriate clock to suit this need, but we have to handle the task of making sure the time always goes forward (especially with multi-processor systems where the TSC might not be synchronised among processors).

The LTTng kernel trace clock for the x86 architecture uses the TSC directly as long as the time keeps incrementing monotonically. If a local TSC is discovered to lag behind the highest TSC counter recorded, the TSC is incremented by an amount lower than the execution time of the check routine (statically defined).

At analysis time, we need to convert the cycles into seconds and nanoseconds, so the frequency of the TSC is required and exported in the trace as part of the `statedump` phase.

4.5.2 LTTng trace clock for user-space

The Linux kernel exports some of its system calls as Virtual Dynamically-linked Shared Objects (vDSO), they are designed to help the user-space execute some kernel actions without the overhead of a system call, and they also provide a fallback mechanism for choosing the most efficient system call mechanism. The vDSO `clock_gettime` retrieves the time for five pre-defined clocks. This call populates a `timespec` structure with seconds and nanoseconds.

System calls is the most common way to call kernel code from user-space. When a process needs the help of the kernel to execute a privileged operation such as opening a file, it sets the system calls parameters in the registers and does an interrupt (`int 0x80`). Then the kernel executes the operation and returns the result back to user-space. This interrupt scheme is costly and needs to be avoided when possible.

Some privileged actions don't necessarily modify kernel data, which means that for certain operations, the user-space only needs to read privileged data and that reading won't impact the kernel code. As of Linux kernel version 2.6.37, the system calls `gettimeofday`, `time` and `getcpu` on the x86 64 bits architecture are available as vDSO. When a process starts, a memory page (`vdso`) is mapped in its address space. For the two time-related system

calls, the kernel updates periodically (with a timer) the read-only information available to user-space, and for the getcpu, it returns the scheduler local data.

In order to avoid executing a system call for each event recorded in user-space, and still be able to synchronise at analysis time kernel and user-space traces, we implemented a new clock source and exported it via the `clock_gettime` vDSO. This new clock named `CLOCK_TRACE` is an interface for the user-space to the LTTng kernel trace clock. As previously covered, the LTTng trace clock is a wrapper around the TSC on supported architectures. Its role is to add the protection that the time does not go backward. To export it to user-space as a read-only value, we exported a variable that must be checked by the vDSO to know if it can continue reading the TSC directly or if it needs to fallback to real system call to let the LTTng kernel trace clock return a monotonic value. That way, as long as the TSC is monotonic and synchronised among the cores, the user-space application calls the instruction to read it without any system call.

Since the LTTng kernel tracer records cycle values for each event and the TSC frequency at the beginning of the trace to be able to convert back in seconds and nanoseconds, we used the same approach for the user-space tracer LTTng-UST Pierre-Marc Fournier (2009). As of now, we assume we are running on a machine with processors that keep a constant TSC frequency, if it is not the case, it might be impossible to convert the cycles in seconds. At the beginning of the user-space trace, we require the tracer to execute one system call to retrieve the TSC frequency that is only accessible from the kernel. In order to do so, we implemented an other clock source in the `clock_gettime` vDSO named `CLOCK_TRACE_FREQ` which executes a system call to return the TSC frequency (measured at boot time).

On supported systems, with the frequency and the cycle counter value recorded in both kernel and user-space traces, we are able to merge with minimal overhead the two traces at analysis time, which gives us a consistent view of the two layers. When the application (such as QEMU) is instrumented with LTTng-UST, it gives the administrator and the developper a high level of granularity in the tracing with the lowest overhead possible because the synchronisation cost is non-existant while recording and the two tracers (kernel and user-space) only need system calls periodically to flush the buffers to disk.

On systems where the cycle counter is not synchronised across CPUs, or the TSC frequency is unstable, we need to fallback to a degraded mode. This mode still gives a synchronised view of the two layers, but it uses a slower and less granular clock source (the chosen kernel clock source).

4.6 Results

4.6.1 QEMU and KVM trace synchronisation

In this scenario, a Linux VM is booting up while we record the trace of the QEMU process in user-space and all the LTTng markers in kernel space. The following listing is an excerpt of the output produced by LTTV in text-only mode when it parses the kernel and the user-space trace. For clarity we only display the events happening on the same CPU and we kept only the relevant fields.

```
ust.cpu_in: 914.540034676,
    { addr = 0x92, value = 0 }
kernel.syscall_entry: 914.540034988,
    /usr/bin/kvm, SYSCALL
    { ip = 0x7f2127dbfa47, syscall_id = 16
        [sys_ioctl+0x0/0xe0] }
fs.ioctl: 914.540035288,
    /usr/bin/kvm, SYSCALL
    { fd = 59, cmd = 44672, arg = 0 }
kvm.kvm_entry: 914.540036082,
    /usr/bin/kvm, VIRT { vcpu = 0 }
kvm.kvm_exit: 914.540037373,
    /usr/bin/kvm, SYSCALL
    { reason = 30 [io_instruction] }
kernel.syscall_exit: 914.540038688,
    /usr/bin/kvm, USER_MODE { ret = 0 }
ust.cpu_out: 914.540039810,
    { addr = 0x92, value = 2 }
```

We can see that the QEMU user-space event `ust.cpu_in` is hit right before an `ioctl` which asks the hypervisor to execute natively the VM code (`kvm_entry`) on virtual CPU 0. Then, a physical CPU is executing code inside the VM and it exits, the `ioctl` system call returns and the `ust.cpu_out` event is hit. From this information, we can compute that even though the `kvm` thread was running for 5134 nanoseconds, the actual VM runtime was 1291 nanoseconds. This information is automatically aggregated in `LTTV` statistics module.

From this listing we also see the LTTV state system working, going from `USER_MODE` to `SYSCALL` to `VIRT` and back.

As we can see, no external event was necessary to synchronise the two traces (kernel and user-space), they were recorded independently and merged at analysis time because they share the same accurate clock source, and our hardware has a constant TSC frequency (Intel(R) Core(TM) i7 920).

4.6.2 User-space trace clock benchmarks

The purpose of this benchmark is to measure the performance and scalability for user-space tracing of LTTng-UST with the new trace clock. Each thread generates 10 million events, the number of threads varies from one to eight. Each event generates a time-stamp and contains a 4-byte integer value as payload. The function writing the events is called in loop. We run this benchmark on a 8-core Intel Xeon E5405 (two quad-core) at 2.0GHz, 16GB ram running Linux 2.6.37 (custom build, debuginfo enabled and LTTng trace clock available).

Running the test program without any instrumentation gives us the baseline of 0.33 seconds for 1, 2, 4 and 8 threads. The table 4.1 displays the time of completion of the test program with the new traceclock (w/ TC) and with the Linux generic clocksource (w/o TC), in both case the baseline has been substracted so the time is the tracing overhead :

Tableau 4.1 UST benchmark

Threads	UST w/ TC	UST w/o TC
1	0 :01.81	0 :02.25
2	0 :01.86	0 :02.13
4	0 :01.86	0 :02.22
8	0 :01.97	0 :02.14

We see that the average overhead of an event is 218.5 nanoseconds without the new trace clock and 187.5 nanoseconds with it. From these results we also see that the new clock scales linearly with the number of concurrrent processes and we obtain an average of 16.25% of speed improvement from reading the time with the default kernel clock source.

4.6.3 Visual Representation

One particularly interesting aspect of adding the new state to LTTV is the visual feedback it provides when using the graphical interface. In addition to the statistics and accurate measuring of the time spent in the VM, we can also graphically see the time spent in each state. For example, in figure 4.1, we see five KVM processes (one parent and four virtual CPUs) switching between VIRT, SYSCALL, WAIT-CPU, WAIT-IO.

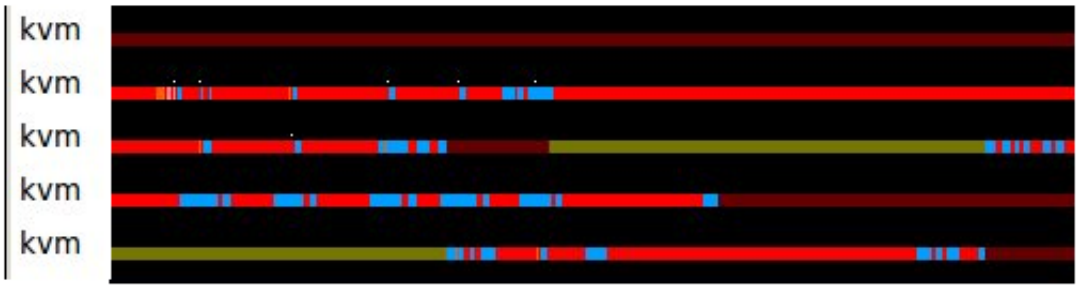

Figure 4.1 Visual representation of a the VIRT state

An other interesting graphical representation is the task migration. In figure 4.2, we see two KVM running a CPU intensive task inside the VM (an busy loop), one task remains uninterrupted but the second one gets migrated while running from one physical CPU to an other. If we dig deeper inside the trace, we can see that just before the task is migrated, an interrupt is received on this processor, so the task running is preempted and the scheduler migrates it to run on an less busy processor. This information is particularly useful for validating constraints such as CPU pinning and also measuring the time lost while the task got migrated.

Figure 4.2 Visual representation of a KVM process migrating to an other CPU

4.7 Future Work

Now that we are able to trace the user and kernel-space and analyse the traces, the next step is to link the traces recorded inside the VM and the host. For the simple case, it has already been done as a prototype for this research, but handling the VM pause and migration poses a significant challenge. Time virtualisation is costly, often inaccurate and a VM trace can contain time gaps only visible from the host point of view, which means that in order

to synchronise the traces at analysis time, we need to synchronise the traces collected on all the hosts M. Jabbarifar et Shameli-Sendi (2011) involved in the VM lifetime, a way to handle all the cases where the `TSC_OFFSET` changes and also all the TSC frequency changes that can occur during this time.

4.8 Conclusion

In this paper we introduced a new state to represent at analysis time a processor running virtual machine instructions and presented the new user-space trace clock, that allows to merge traces recorded inside the kernel and in user-space at analysis time. These modifications are now merged inside the LTTng codebase and they provide an efficient way to record and analyse traces generated from the KVM hypervisor. The whole concept is easily extensible to other user-space programs, interacting with kernel-space components and to other hypervisor, provided the instrumentation effort is done.

Chapitre 5

LINUX KERNEL TRACING AS A PROBLEM-FINDING METHODOLOGY

Authors

Julien Desfossez
École Polytechnique de Montréal
julien.desfossez@polymtl.ca

Michel R. Dagenais
École Polytechnique de Montréal
michel.dagenais@polymtl.ca

Submitted to

Journal of Computer Networks and Communications

Keywords

Linux, Tracing

Abstract

Many tools and methodologies exist for identifying and solving problems in Linux. Most of the time, when something wrong occurs on a system, firing tools like `top`, `vmstat` and others for a few seconds and reading log files is enough to find the culprit. But the worst enemy of a system administrator is the bug that happens sometimes, leaves no trace and disappears by itself, until its next appearance. In this paper, we present how a low-impact

tracer such as LTTng, combined with a proper methodology, can help identifying and solving this class of problems. For that matter, we will go through the process of analysing a problem with trace data and then introduce a new trace analysis tool, `LTTngTop`.

5.1 Introduction

General purpose operating systems such as GNU/Linux come with tools to identify problems related to abusive resource usage. Most of the time, the statistics provided by these tools are enough to identify crashed applications and resources shortage. For large infrastructure, these statistics are often gathered and centrally displayed to allow system administrators to have a quick overview of their data center. This method, combined with a database to store the data over time, is called trending and is particularly useful to see the evolution of resource usage.

When an application behaves wrongly, and this behaviour is not related to a specific resource exhaustion, it is often possible to activate log file creation with various level of informations. If the application is well designed, these log files should provide enough information for a system administrator, or at least a developer, to understand what is happening inside the application. Of course, increasing the log level also increases the load on the machine, and might contribute to create other problems. If the problem seems to be network related, a packet sniffing tool such as `tcpdump` may be useful.

When high level information such as statistics, trending, log files and eventually network packet analysis are not enough, looking at the system from a kernel point of view is usually the next logical step. If the faulty application is known, and the problem is happening while the administrator is connected on the machine, it is possible to use the system call `ptrace` with tools such as `strace` or `gdb`, it will have an important impact on the performance of the system, but it might give enough information.

This methodology is usually a good practice to solve most common problems but, when it is not, depending on the importance of the problem and the time allocated to solve it, system administrators often end up with *ad-hoc* solutions. For example, when a bug appears and disappears randomly, or when it is related to complex interactions between processes, and the only information available is a feeling that the system is sometimes too slow, it is a really hard task to find the source of the problem. The same applies for complex concurrency optimisation : there are no tools or methodologies to record enough information on production systems and then analyse it efficiently.

In this paper, we present in section 5.2 work related to this particular area of problem solving, in section 5.3 the family of problems we are addressing with kernel tracing and how

we handle them, in section 5.4 we introduce the tool `LTTngTop` and how we can use it to sort through the amount of data generated when recording a long trace containing a problem. In section 5.5, we show the results of the benchmark measuring the overhead of load that `LTTng` and `LTTngTop` add on the system. Finally, we will talk about possible future work in section 5.6, and conclude in section 5.7.

5.2 Related Work

In Ruan et Pai (2004), the authors struggled against a similar type of problem : the lack of non-intrusive and detailed instrumentation. Their objective was to optimize the performance of the `SpecWeb99` benchmark and they realized there was no tool to detail accurately every call that was fast, detailed and could work while the program was running. Most of the tools and statistics indeed use sampling and provide average numbers, which is a problem when we want to understand a specific case where the problem occurs. As a result, they developed `DeBox` for the `FreeBSD` kernel. This allows the application developers to retrieve performance information about each system call. Instead of only returning the system call error code, they also return a structure where the application developer can find detailed performance data related to the system call they just executed. That way, they know precisely and for each system call the impact on the system. Thanks to this tool, they were able to dramatically improve the performance of the benchmark on their system and improved a system call in the `FreeBSD` kernel. This design is different from tracing as we intend to implement because it is targeted for direct support in the application being traced : the application is in charge of reading the performance information and eventually react to the information provided, which is different from system-wide tracing that helps us understand complex interactions between processes.

The paper Sushant Sharma et Maccabe (2005) is an other project where detailed syscall performance is an issue. The authors worked on optimising the system call overhead for high performance computing applications. For that matter, they enhanced the `Linux Trace Toolkit` to extract performance information from the linux kernel, as part of the `system call return` event. That way, like in the previous paper, when a system call returns, they gather detailed information and have a better understanding of what is causing the latency. Once again, we see that only knowing a system call occurred is not enough for optimisation, there is a need for tools with a low overhead that provide performance information related to each event. In our case, we use the performance counters to provide this information in a more generic manner.

5.3 Kernel Tracer for Problem-Solving

5.3.1 Categories of problems

With kernel tracing, we obtain a complete overview of the operating system. From there, we can study very closely the interactions between the various subsystems and, with the proper tools, extract the relevant information. The problems we are addressing here are those based on complex interactions between processes, the kernel and peripherals. For example, we look for unexplained latency in the execution of a program and concurrency issues.

5.3.2 Requirements

One particular aspect to take into account, when dealing with latency or concurrency problems, is the impact of the tracer on the problem itself. For example, when debugging a thread synchronisation problem, adding an instruction to output debugging information might be enough to completely change the order of execution of instructions and give the impression that the problem is solved, or at least different from what it really is.

Since we are targeting problems happening on production systems, the tracer we choose must be entreprise-grade, which means that it can be deployed in the context of enterprise servers. This characteristic does not only depend on the efficiency or accuracy of the tracer, the focus of most research. Besides limiting the impact on the system and the traced applications, the tracer must also have the granularity to only enable specific targeted events, be operated by non-root but trusted users, support concurrent tracing sessions, have a security scheme that complies with production servers, and have all the associated tools to efficiently control and analyse the traces in a console-only environment.

5.3.3 Known solutions

When looking at mature Linux kernel tracing solutions, the choice is composed of four tools : `ftrace` Edge (2009a), `LTTng` M. Desnoyers (2006), `perf` Edge (2009b) and `SystemTap` Vara Prasad (2005).

The two tracers integrated in the mainline kernel, `ftrace` and `perf`, both provide efficiently some really interesting data that can be used to find the cause of some problems, but their main focus (in both cases) is developers, especially kernel developers.

First of all, `ftrace` is the kernel function tracer, it was designed at the beginning to generate events at the entry and exit of kernel functions. Since then, various components have been added to make it able to perform various analysis related especially to latency and scheduling issues. One big advantage is that it is a pure kernel tracer, so there is absolutely

no need for external user-space tools to control it and view a textual representation of the traces, everything is done through the `debugfs` interface. Of course, to sort through the amount of data generated, a text output is hardly enough, so a graphical interface also exists (KernelShark) and provides some interesting features such as zooming and filtering the events. The main problem with `ftrace` for our particular case is that it is mostly designed for developers which means that the user needs to be the administrator of the server, have a good understanding of the kernel operations, be the only one using the tracer, and ideally be able to reproduce the problem quickly and with no particular load, otherwise the relevant information might get lost in the noise.

The tracer `Systemtap` is specifically targeted for system administrators. It is a high level interface to `kprobes` Rostedt (2005) and its goal is to quickly allow the user to create scripts to collect data from a kernel sub-system. Although the `SystemTap` script language is relatively high level and provides interesting data structures, it requires knowledge of the kernel behaviour. Indeed, each script is compiled as a kernel module and loaded upon execution. When it is loaded, it gathers informations and outputs it when necessary to the console or a log file. Thus, we need to have a good understanding of the kernel and usually read the source code to know which variable to gather. Moreover, `SystemTap` does not provide a binary trace format, which means that the result is in a text file that grows rapidly and adds a significant load to the system.

5.3.4 LTTng

LTTng, the Linux Trace Toolkit Next Generation, is a highly efficient full system tracing solution toolchain. It is composed of several components to allow tracing (kernel and user-space), trace viewing, analysis and streaming. It started around 2006 as an out-of-tree kernel patch and was one of the first efficient kernel tracer. The new version, LTTng 2.0, is now focusing on the usability for end users. The entire core is based on modules instead of kernel patch, which means that it is possible to use LTTng on vanilla and distribution kernels, starting with version 2.6.35. Moreover, the tools to control the tracer and read the traces are now focused on the enterprise requirements, so a tracing group is allowed to control the tracer (not just root), and multiple users can record different traces with different instrumentation sets. Moreover, the kernel data source available to LTTng include tracepoints, the function tracer, the CPU Performance Monitoring Unit counters and kprobes support. This new version of LTTng produces an output in the Common Trace Format (CTF), format which aims to become a standard format for tracers.

For this work, we chose to use the LTTng tracer, since it seems to be more suited for our

requirements.

5.3.5 Overview of problem solving with LTTng

When trying to figure out what is causing a performance or latency problem on a machine, the first step is usually to have high level information about the source of the problem. If we are down to using a kernel tracer, chances are that the problem is not easily solvable by looking at the log files or tools like `top`. Thus our first step is to record enough information to get a general idea of where the problem is coming from, without impacting too much the system. If the problem is reproducible, it will be easier since we will only record for the time required to trigger the problem. In the following sections we will see the types of information we can retrieve with LTTng.

Tracepoints

The tracepoints are defined by the `TRACE_EVENT` macro inside the kernel Rostedt (2010) as a source of kernel information. This macro is used throughout the kernel source code to allow the various kernel tracers to attach and retrieve relevant and selective information about the kernel activity. In order to do so, they have to attach `probes` which are called when the event is hit. In LTTng 2.0, once the probes are created, they are activated with the `enable-event` command. The list of available tracepoints on the running kernel can be retrieved with the `list -k` command.

For example, if we want to record the `irq_handler_entry` and `irq_handler_exit`, we run the following command

```
$ lttng enable-event -k \
 irq_handler_entry,irq_handler_exit
```

The following output shows the textual representation of two recorded events :

```
timestamp = 49251984081970,
 name = irq_handler_entry,
 stream.packet.context = { cpu_id = 0 },
 event.fields = { irq = 0, name = "timer" }
timestamp = 49251984097754,
 name = irq_handler_exit,
 stream.packet.context = { cpu_id = 0 },
 event.fields = { irq = 0, ret = 1 }
```

The timestamp is the number of nanoseconds elapsed since the boot of the machine. With this information, we know precisely that the IRQ handler was active for 15.7 microseconds to handle the timer interrupt.

Moreover, tracepoint information is also useful to follow the code flow, for example when activating the system call events and more internal tracepoints such as filesystem and block device tracepoints, we see the chain of events, starting by the system call going down to the virtual filesystem interface (VFS) and to the block device, receiving the interrupt and going back all the way to user-space. All of this information is interesting when we are dealing with latency issues and we want to explain some disturbing delays.

Statistics with the Performance Counters

Depending on the resource impacted by the problem (network, disk, memory, cpu), collecting statistics is usually a good way to narrow down the problem. As explained above, LTTng 2.0 can read and export the performance counters everytime it records an event. There are a many different software and hardware performance counters available on modern architectures, the list supported by LTTng is available in the help menu of the `add-context` command. The following listing is an excerpt of the output of this command :

```
$ lttng add-context -h
perf:instructions, perf:cache-references
perf:cache-misses, perf:branch-misses,
perf:branch-instructions, perf:branches,
perf:L1-dcache-loads,
perf:L1-dcache-load-misses,
perf:dTLB-load-misses, perf:page-fault,
perf:faults, perf:major-faults,
perf:minor-faults, perf:context-switches,
perf:cpu-migrations
...
```

This information is represented as a context information for all or for specific events, so they need to be attached to the events we are interested in. This is particularly useful when we want to see precisely the impact of a particular event on a counter. For example, if we want to know the number of major page faults generated by a process, we record the scheduling events (`sched_switch`) and we attach to these events the `perf:major-faults` context. Everytime a process is scheduled out, we will extract the value of this counter, so

between two scheduling events, we know precisely the difference between the two values, which gives us the amount generated during the time the process was running.

The following example shows this analysis with a simplified output generated by the converter `babeltrace`, we see that when the process `soffice.bin` is scheduled in, the major faults counter is at 20, and 3.2 milliseconds later when it is scheduled out, there has been one new major fault. So we know that during that time, the process generated a major page fault which triggered a reading on the hard drive.

```
516162551567239, sched_switch,
    perf_major_faults = 20,
    prev_comm = "kworker/0:3",
    next_comm = "soffice.bin"
516162554766321, sched_switch,
    perf_major_faults = 21,
    prev_comm = "soffice.bin",
    next_comm = "awesome"
```

Digging more in depth with ftrace and kprobes

After carefully using the tracepoints and performance counters, we should have a good idea of where the problem is coming from. For most problems occuring because of a user-space malfunction, it should be enough to know why it is happening. If the problem is identified as a kernel malfunction, we might need more detailed information not provided by the common tracepoints : we need to customize the registered events. For that matter, LTTng is able to interface with the function trace (also known as `ftrace`), and for even more fine grained information with `kprobes`.

5.4 Tracing in the Real World

In the previous section, we detailed some of the features LTTng provides as a kernel tracer and how it is useful for problem solving, but we need to keep in mind that this information is highly accurate and high throughput. For example, when only recording events related to IRQ handler entries and exits on an idle dual-core machine for three seconds, we end up with a trace containing more than 2500 events. When recording all the system calls on the same machine for the same amount of time, we end up with more than 2300 events. This number of events is far too high to allow a system administrator to find quickly the relevant information, allowing him to solve a problem.

Moreover, when the problem appears at random under difficult to reproduce circumstances, we need to be able to record a trace for the necessary duration before the problem appears. Thus, we could be recording a trace for a long period of time. Manually digging into such a trace, after we stop the recording, is nearly impossible or consumes a lot of time. In this section, we present the tool `LTTngTop`, designed to solve this particular issue and allow system administrators to use tracing as a problem-solving tool.

5.4.1 LTTngTop

LTTngTop is a console-only tool. Its main purpose is to provide system administrators with a convenient way to browse through traces and find quickly a problem, or at least a period of time when it happened. From this information, we considerably reduce the number of events we need to analyse manually. It is designed to suit the system administrators because it behaves like the popular `top` CPU activity monitoring program. In addition to the usual behaviour of `top` and similar tools, where the display is refreshed at a fixed interval, `LTTngTop` allows the user to pause the reading of the trace to take time to look at what is happening, and also to go back and forth in time to see easily the evolution between two states.

In order to properly handle the events without the risk of loosing coherence, and attributing statistics to the wrong process in case of lost event, we require that the events are self-described. Using the context information, it is required that each event includes the process identifier (`PID`), the process name (`procname`, the thread identifier (`tid`) and parent process identifier (`ppid`). Although adding this information makes the trace bigger, it ensures that every event is handled appropriately, even if we loose some events (which can happen if the trace subbuffers are too small).

Displaying the trace offline

It can be used offline or online, which means that we can use this tool to browse a trace recorded previously, or to display the trace while tracing. The tool does not directly output the trace to the user, like `babeltrace` does. Instead, it shows statistics aggregated from reading the trace, thus showing to the user a summary of the activity on the machine during a period of time. Like `top`, it refreshes the display at a certain interval of time (by default one second) showing only a window of the trace which can be used later to limit the scope of manual analysis of the events. As of now, the statistics displayed are the CPU usage time, the performance counters and bandwidth of inputs and outputs (on both network and disk).

Displaying the trace online

`LTTngTop` is useful for browsing a trace and finding specific periods of time where something suspect occurs, but it can also be used while the trace is being recorded. The `LTTng` kernel traces are usually consumed by a user-space daemon with the `splice` system call : when a buffer is full, the consumer is notified via the return of the `poll` system call, it reads the data and writes it to disk. In this scenario, the `splice` system call is used to read data from a file descriptor to a pipe, without copying it intermediately, that way the writing to disk is efficient. However, it is also possible to consume the data directly inside a memory mapped region. In this mode, every time we want to consume the data, we ask the kernel via an `ioctl` to map a trace subbuffer in our address space. In `LTTngTop`, since we want to refresh the display at a fixed rate, we cannot wait for the `poll` system call to return, so we use the snapshot mechanism available. Everytime we want to refresh the display, we ask the kernel tracer to snapshot the state of each active stream (usually one stream per cpu) and then we read the data from the last snapshot to the new ones just acquired. During that time, the kernel tracer is writing new events in other subbuffers.

When in online mode, the trace is not necessarily written to disk (the user choose to start writing to disk if necessary), which means that the impact of the tracer is really minimal. Since we want to still allow the user to go back in time to see what happened, at every period we record the state of the system in memory. On a typical desktop system, each state represents around one hundred bytes of memory that is (for now) never freed while the trace is running. Future plans include an option to configure the size of the history wanted.

The mechanism to pause the display is also available in online mode. When the user requests a pause, we stop the recording of the trace and pause the display. That way, the user can review the previous states and the tracer does not impact the system. Since we don't know the time of the pause, we chose not to record the trace and compute the statistics while in pause. When the user wants to start again reading the trace, it resume and the tracer starts collecting data with the same parameters. Of course there is time gap while the system is in pause, but we don't loose coherence since each event is self-described.

CPUTop

The CPUTop view shows the CPU time usage of each process for the last time period. Instead of relying on the kernel statistics, which are only available while the process is alive, we record the scheduling events and know precisely when a process is running and when it is waiting for the processor.

When using `LTTngTop` in live mode, it can be seen as a replacement for `top` because it

provides the same information but is less intrusive in terms of system calls. Indeed, at every refresh, `top` browses the `procfs` pseudo-filesystem to retrieve the list of live processes, and then, for each process, it opens, reads and closes its `stat` file. Although this tool is often considered as essential on every Unix-based system, it is intrusive because it generates three system calls for each process and then displays the results. With `LTTngTop` in online mode, we read the trace data from a memory mapped region. We only use system calls to inform the kernel tracer we want a snapshot and to map the memory region. Then, we read the trace data directly from this region. Depending on the trace buffer size, the amount of information in a snapshot varies. The benchmark of the overhead of CPU load is available in section 5.5.

PerfTop

In a previous section, we saw that it is possible to use the performance counters as context information for LTTng events. We also saw that it is possible to use this information manually. Nonetheless in order to have an overview of what is happening as a whole on the system, we need an automated way of computing and sorting this information per process. This view does exactly this task : it shows to the user the various performance counters enabled during the trace and allows to display and sort them. Like the CPUTop view, the data is reset to zero at every new period.

IOTop

Thanks to trace data, and particularly system call events, we know every input/output generated by every process. With this information, we can aggregate for a certain period of time the `read` and `write` operations and display to the user the network and disk bandwidth for each process. This is really useful to find input/output related problems.

As of now, the level of details varies depending on the events recorded. If we record a trace and a file descriptor is already open (either by a socket creation or a file open), we cannot distinguish network or disk operations since they both use the same system calls. In order to solve this problem, a dump of the opened file descriptors of every process should be gathered when the trace is starting, but this operation is not yet implemented.

Sometimes, knowing that a process is doing a lot of I/O is not enough. For example, with a file server such as `Samba`, it could be useful to know more than just the bandwidth. This is why the IOTop view provides a way to display for each process the bandwidth associated with every open file descriptor and, if available, the name of the file or the adress connected to the socket.

5.5 Results

One particularly important feature of a tracer, besides its accuracy and the data it produces, is its impact on the traced system. Since we are working on production servers and our users are system administrators, the benchmark presented here are comparisons of intrusiveness between `LTTngTop` and `top`. The first benchmark is done with sysbench to measure the difference of impact on CPU-intensive tasks. The second one is using the `LTTng` kernel tracer to measure the number of system calls made by the programs and the total cpu time.

5.5.1 sysbench CPU

In this section we present the benchmark realized to measure the overhead of CPU load added by the tracing solution. The benchmark consists of comparing the time of completion of the CPU performance test from `SysBench`. In this test, each request consists in calculation of prime numbers up to 10000. We launch two threads in parallel on a dual-core system. Each test is run 10 times under the same condition. At the end, we compute the average and the standard deviation. The following table shows the results, with the test running without any tracing (alone), with `LTTng` tracing to disk, with `top` running at a refresh rate of one second and with `LTTngTop` running in online mode (parsing the trace live without writing it to disk) with a refresh rate of one second. The first line of the table 5.1 is the average time of completion of the 10 tests and the second line is the standard deviation.

In the tests with the kernel tracer, it is configured with four subbuffers of one megabyte. We record the `sched_switch` events with the `pid`, `tid`, `ppid`, `procname`, `perf:cache-misses`, `perf:major-faults`, `perf:branch-load-misses` context information collected for each event.

For the tests with a display refresh, since most of the time is spent redrawing the window, the console size is the same : 87x57 characters.

Tableau 5.1 LTTngTop benchmark

Alone	LTTng	top	LTTngTop
5.31837	5.32372	5.34395	5.34204
0.00417	0.00637	0.00739	0.00702

As we can see, the overhead added by the tracing solution in terms of CPU usage is very small and arguably a little less than the load imposed by `top`. Moreover, the results with the tracing are stored in memory (or disk) and can be consulted again at any time, and the data collected is more complete than the one produced by `top`, since we read three performance

counter values while tracing.

5.5.2 Measurement with tracing

In this test, we started in two separate terminals `top` and `LTTngTop` with the same window size and a refresh rate of one second. We also started a `LTTng` kernel trace recording the system calls and the scheduling events and recorded the trace for three minutes. The two programs were started and stopped outside the scope of the trace, so we only recorded while they were running.

In terms of number of events and number of system calls, the results are presented in table 5.2.

Tableau 5.2 LTTngTop vs top cpu number of events

	top	LTTngTop
Number of events	493018	28691
Number of syscalls	241933	13981

The system calls executed by `LTTngTop` are presented in table 5.3

Tableau 5.3 LTTngTop number of system calls

count	Syscall name
6413	sys_mprotect
3043	sys_ioctl
1627	sys_poll
1505	sys_rt_sigaction
482	sys_write
374	sys_futex
358	sys_rt_sigprocmask
179	sys_nanosleep

The system calls executed by `top` are presented in table 5.4

Tableau 5.4 top number of system calls

count	Syscall name
66014	sys_read
63198	sys_open
63198	sys_close
31599	sys_newstat
6336	sys_alarm
5104	sys_fcntl
4224	sys_rt_sigaction
880	sys_lseek
352	sys_ioctl
352	sys_getdents
324	sys_write
176	sys_select
176	sys_access

The total CPU time of the two programs is presented in the table 5.5

Tableau 5.5 LTTngTop vs top total CPU time

	LTTngTop	top
Total CPU time	0.884s	2.408s

From this test, we see that `LTTngTop` consumes 272% less CPU than `top` under the exact same testing conditions and provides a lot more information. Part of this result might be related to the way top displays its information, but judging from the amount of system calls (the trio open, read and close for each process) executed by `top`, we can say that `LTTngTop` is less intrusive.

5.6 Future Work

The `LTTngTop` tool allows the user to quickly get statistics about the system where the trace was recorded. Since this tool is designed for system administrators, the next step is to enhance the statistics generated, add more relevant information that can help solve problems quickly. Once the data for one machine will be considered sufficient, the tool will evolve to become a network console where it can read statistics coming from a complete data center. That way, it will allow a system administrator to quickly have an overview of what is happening, and sort the data based on high level informations such as processor, disk, memory and network usage and then dig inside specific servers to get more accurate

informations.

While doing this research, it was noted that the desktop widget displaying the load on desktop machines are actually really costly. When looking at the statistics collected by `powertop` on a Gnome Desktop with the CPU load widget activated, we noticed it was the first cause of wake up whereas `LTTngTop` did not even appear. We intend to create a library that could be used by such applications to read the statistics it needs directly from the tracing core instead of this traditionnal method of opening and parsing files.

5.7 Conclusion

In this paper, we presented a methodology to make tracing useful for system administrators. The `LTTng` tracing toolchain is enterprise-grade, provides highly detailed informations about the activity of the operating system and, with `LTTngTop`, it is easy to have a higher level overview of statistics. These statistics are gathered from the events but also from the performance counters. We also demonstrated that the overhead of tracing when targeted for a specific usage is really small, provides more detailed information and is more efficient than common tools based on polling.

Finally, we have a strong belief that tracing, when configured properly, could be the base of every tool requiring access to kernel-level statistics. The overhead is small, and instead of requiring context switches and kernel locks (which is what happens when we access a `stats` file in `procfs`), we read the data collected in a memory-mapped region.

Chapitre 6

DISCUSSIONS GÉNÉRALES

Dans cette section, nous allons revenir sur les résultats présentés dans les articles, puis discuter l'impact de ces contributions et détailler les limitations des solutions présentées.

6.1 Retour sur les résultats

Tout d'abord au niveau du premier article [4] et de son implémentation, la première contribution apportée se situe au niveau de la méthode pour synchroniser des traces à plusieurs niveaux. En effet, l'approche prise par ce travail démontre qu'il est possible d'utiliser une source de temps très granulaire telle que le TSC comme base de temps de référence lorsque les tests de consistance sont réalisés efficacement. L'implémentation et les résultats montrent, qu'avec les résultats des tests de consistance exportés de manière efficace en espace utilisateur, il est possible d'utiliser le TSC tel quel à plusieurs niveaux et ainsi obtenir des traces synchronisées au plus bas coût possible.

Le premier article présente également le nouvel état CPU `VIRT` qui présente à l'utilisateur une nouvelle représentation de l'utilisation des processeurs. En effet, les processeurs virtuels étant gérés par des processus en espace utilisateur sur la machine physique, il est important de différencier quand un de ces processus fait un appel système et quand cet appel système consiste à réellement exécuter du code de la machine virtuelle. Avec ces résultats, l'analyse de trace de machine virtuelle devient plus claire car il est possible de mesurer très précisément le temps passé à réellement exécuter du code.

Dans le second article [5], la contribution se situe au niveau de la méthodologie pour extraire des métriques systèmes depuis les traces noyau, et au niveau de la manière de les utiliser pour des administrateurs systèmes. Nous présentons ici que les métriques systèmes, acquises par la lecture périodique de compteurs du noyau, ajoutent de la contention sur le système. Nous pouvons aussi ajouter que cette méthode d'échantillonnage ne permet pas d'obtenir des informations précises sur l'utilisation réelle du système, contrairement au traçage qui exporte toutes les données possibles. Il devient donc important de trouver un équilibre entre précision et charge.

6.2 Limitations de la solution proposée

L'enregistrement de traces à plusieurs niveaux pour l'instant ne prend pas en considération la migration éventuelle des machines virtuelles. Lorsqu'une machine virtuelle change d'hôte physique, ou lorsqu'elle est mise en pause, le temps s'arrête dans celle-ci et continue lorsqu'elle est de nouveau fonctionnelle. Cette arrêt virtuel du temps est simulé en appliquant un décalage sur l'horloge du système. Cette particularité rend la synchronisation des traces particulièrement difficile et la recherche pour contourner ce problème implique une synchronisation des traces entre les machines physiques et une prise en considération de la particularité qu'un système d'exploitation peut avoir des périodes d'inactivité sans pour autant que la machine soit arrêtée.

Dans le travail dédié à la recherche de problèmes dans les environnements de production, la solution proposée manque pour l'instant d'un système pour stocker les différents états d'une machine. L'ajout de ce composant permettrait d'améliorer les capacités de recherche et ainsi de créer de meilleures métriques basées sur l'évolution des métriques plutôt que sur des données instantanées.

6.3 `LTTngTop`

Un des résultats de cette recherche est la création du logiciel de visualisation de traces `LTTngTop`. Tout comme le populaire outil `top`, il affiche des informations dans une console et rafraîchit les données présentées à intervalle régulier. Cet outil est conçu tout spécialement pour les besoins des administrateurs systèmes qui travaillent sur des systèmes distants accessibles seulement par SSH et sur lesquels il doivent diagnostiquer avec le moins d'impact possible les éventuels problèmes.

6.3.1 Modes de fonctionnement

`LTTngTop` a deux modes de fonctionnement : avec des traces déjà enregistrées ou pendant la capture. Dans le cas où les traces sont déjà enregistrées, il sert de navigateur dans la trace, les flèches du clavier permettent de contrôler la navigation, il est possible de laisser défiler la trace à la même vitesse que cela s'est produit sur la machine ainsi que de revenir en arrière et mettre en pause.

Dans le cas où l'on ne charge pas de trace déjà enregistrée, le logiciel se charge de créer une session de traçage, active les points de traces intéressants et lit les données de traces directement depuis des zones de mémoires sans jamais écrire la trace sur le disque dur. Dans ce mode, il est également possible de mettre la trace en pause et de revenir en arrière

pour consulter les états précédents. Si on désire également enregister la trace pendant qu'on l'affiche, il est tout à fait possible de créer une autre session de traçage en parallèle. Ainsi il sera possible de creuser plus en profondeur et regarder les événements plus en détail au besoin.

6.3.2 Données présentées

Données génériques

Les données extraites sont des statistiques agrégées pendant une période de temps fixée (par défaut une seconde). Comme on peut le voir sur l'image 6.1, la zone du haut affiche des informations générales sur le système telles que la période de temps concernée dans la trace (ce qui permet plus tard de cibler une zone précise lorsqu'on consulte la trace manuellement), le nombre de processeurs sur la machine, le nombre de processus, le nombre de fils d'exécutions (*thread*), le nombre de fichiers ouverts et le nombre de connexions réseau ouvertes. Les nombres entre parenthèses à côté de ces données sont la différence par rapport à la période d'affichage précédente. Pour les fichiers et les connexions réseau, la bande passante en entrée/sortie est affichée également.

CPUTop

Comme on peut le voir sur l'image 6.1, la zone d'affichage principale, présente par défaut l'utilisation des processeurs par processus, triée par ordre décroissant. Les données sont similaires à celles produites par l'outil `top`, la différence étant qu'elles proviennent directement des événements de l'ordonnanceur et peuvent ainsi être sauvegardées et reconsultées *a posteriori*.

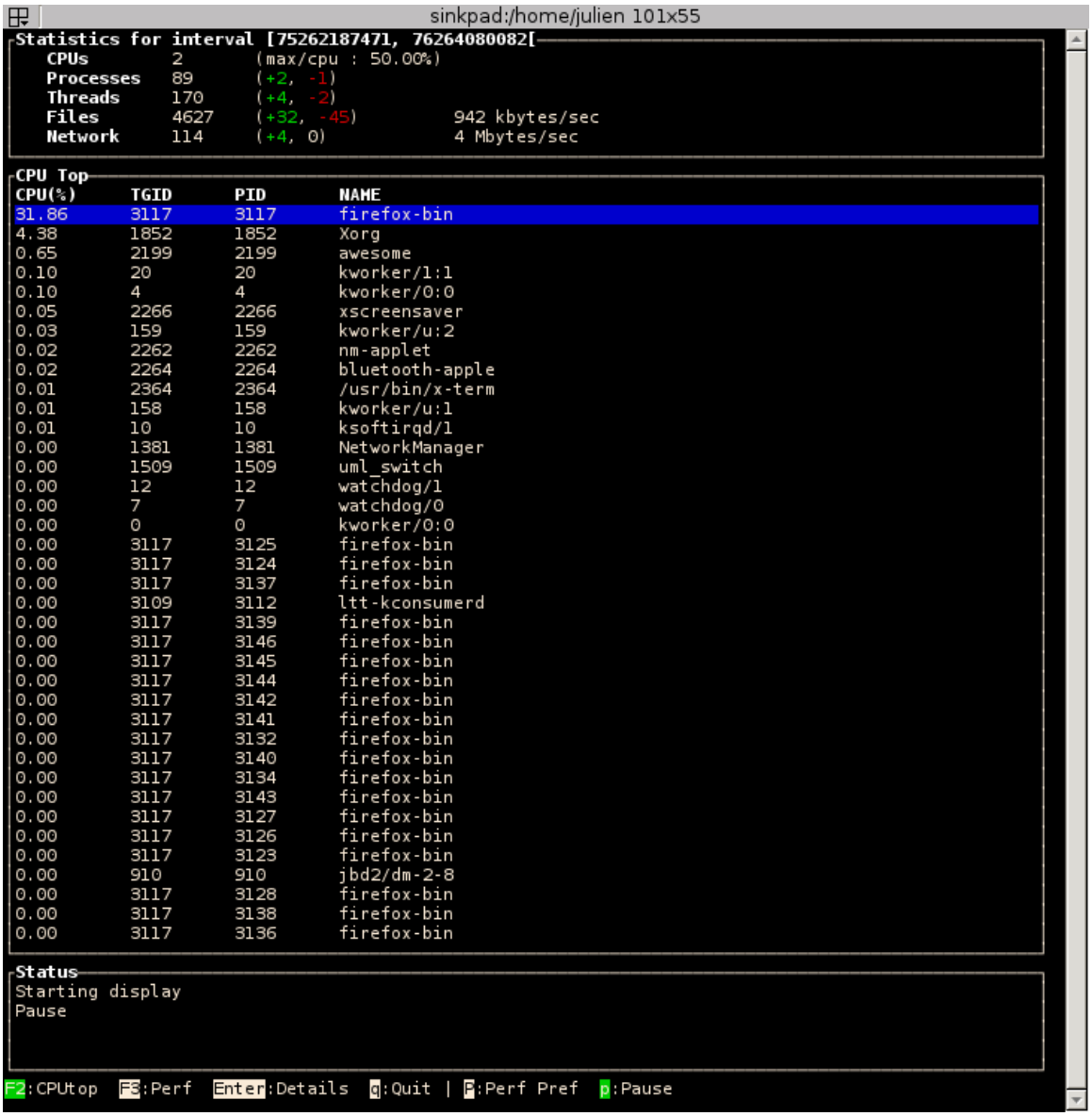

Figure 6.1 Interface CPUTop de LTTngTop

PerfTop

Les compteurs de performance sont également des données très intéressantes, permettant de connaître rapidement le fonctionnement d'un processus et ses intéractions avec les ressources de bas niveau. Ainsi, on peut savoir le nombre de fautes de caches faites par un processus ou le nombre de fautes de pages majeures (qui causent des accès disques). Ces données sont attachées aux événements du traceur noyau `LTTng` sous la forme d'informations de contexte, c'est-à-dire que les compteurs sont lus au moment de l'enregistrement des événements et joints à celui-ci. Traiter l'évolution de ces compteurs manuellement est une tâche fastidieuse, c'est pourquoi la vue `PerfTop` présentée à l'image 6.2 a été créée. Elle affiche l'état des compteurs de performance par rapport à la période en cours et les trie par ordre décroissant. Un menu permet de choisir les compteurs à afficher ainsi que la colonne à choisir pour le tri.

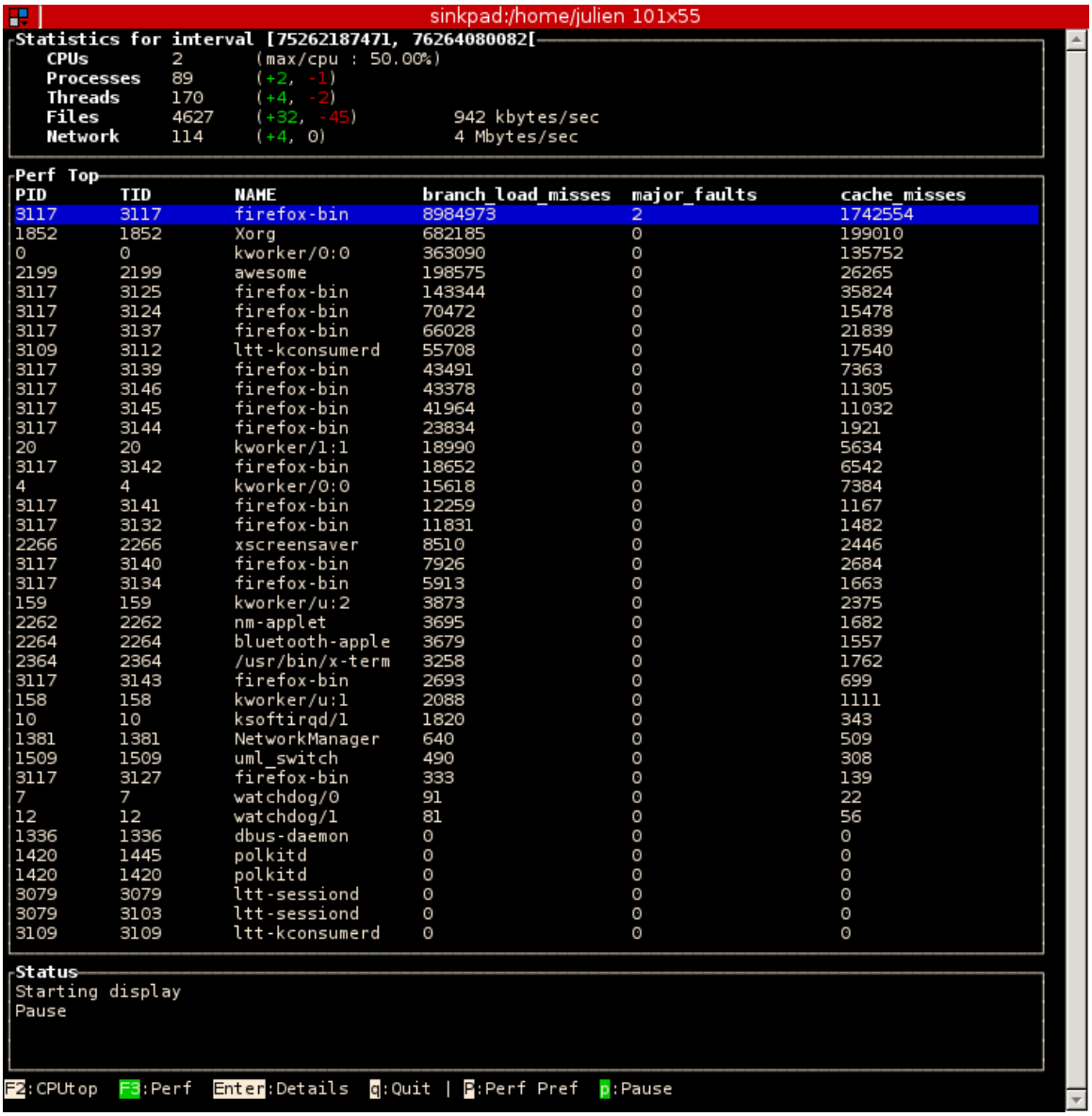

Figure 6.2 Interface PerfTop de LTTngTop

Chapitre 7

CONCLUSION

7.1 Synthèse des travaux

Dans ce mémoire, nous avons étudié la problématique de l'utilisation du traçage pour des administrateurs systèmes dans des parcs informatiques utilisant ou non de la virtualisation. La performance du traceur, la cohérence et l'efficacité du traitement des données sont les critères traités dans cette étude.

La première problématique que nous avons étudiée est la synchronisation des traces enregistrées à différents niveaux du système d'exploitation. L'objectif était d'enregistrer des traces en nécessitant le moins de changements de niveau possible puis, lors de l'analyse, être en mesure de prendre ces différentes traces enregistrées simultanément et obtenir un affichage uniforme et cohérent nous donnant une vue d'ensemble. Cette partie de l'étude nous a amené à étudier les différents mécanismes de synchronisation de trace, les différentes sources de temps disponibles sous Linux sur l'architecture x86, l'impact de traverser des couches dans un système d'exploitation (que ce soit l'espace noyau, utilisateur ou hyperviseur), pour arriver à la solution que nous jugeons optimale. Nous avons choisi d'utiliser le compteur de cycles (TSC) car il s'agit de la source de temps la plus rapide et granulaire accessible sur l'architecture x86 et parce qu'il est accessible à un coût très faible depuis toutes les couches. Toutefois, l'utilisation du TSC seule n'est pas suffisante car il y a de nombreux risques associés, particulièrement sur les architectures à plusieurs processeurs. Les tests pour assurer la cohérence des lectures de temps étant faits dans le noyau, nous avons étudié le moyen de les rendre accessibles à tous les niveaux en ajoutant une nouvelle option au vDSO `clock_gettime`. Grâce à cette solution, le traceur en espace utilisateur `LTTng-UST` et le traceur noyau `LTTng` peuvent enregistrer des traces simultanément sans nécessiter aucune interaction au moment du traçage. Les traces ainsi enregistrées utilisent la même source de temps et peuvent être fusionnées automatiquement lors de l'analyse. De plus, cette nouvelle source de temps a amélioré la performance du traceur en espace utilisateur par un facteur de 16.25%.

Une fois l'enregistrement des traces à différents niveaux rendue possible et efficace, l'étude s'est portée sur l'utilisation de ces traces dans les environnements de production et partic-

ulièrement les centres de données. Dans cette partie de l'étude, de nombreuses contributions ont été apportées à la version 2.0 du traceur noyau `LTTng` afin de rendre son utilisation possible et utile aux administrateurs systèmes. La librairie de collecte des données du noyau (`liblttngkconsumerd`), la libraire d'interface avec le traceur noyau (`liblttngkernctl`) et la création de l'outil `LTTngTop` sont les implémentations de ce projet. Ces outils profitent de la recherche faite en collaboration avec l'entreprise Révolution Linux pour mettre au point une méthodologie et des algorithmes pour aider les administrateurs systèmes à trouver rapidement des problèmes complexes sur un système d'exploitation. Ces algorithmes, principalement basés sur des métriques provenant des points de trace du noyau et des compteurs de performance, ont pour but d'aiguiller les recherches tout en limitant au maximum la charge sur le système. L'outil `LTTngTop` peut ainsi lire des traces *a posteriori* pour extraire des informations de plus haut niveau et aider à cerner un problème. Il peut également être utilisé pour tracer et consulter les traces directement sur le système dans un environnement en console uniquement. Dans les deux cas (avec les traces enregistrées ou affichées directement), les mesures montrent que la charge imposée sur le système est légèrement inférieure à celle imposée par l'outil populaire `top` tout en apportant plus d'informations et de flexibilité que celui-ci. La conclusion de cette recherche est la preuve qu'il est maintenant possible d'utiliser ces technologies et les intégrer aux outils de supervision de parcs informatique.

7.2 Améliorations futures

Pour l'instant, la recherche de problèmes donne des résultats sur une seule machine, l'objectif futur est de l'étendre à un parc informatique complet et bénéficier ainsi des intéractions entre les machines. Grâce à cet ajout, il sera non seulement possible d'avoir les données d'un parc de manière centralisée, mais en plus l'analyse bénéficiera de la connaissance des liens entre les machines, ce qui étendra considérablement la précision des données. Pour l'intégration avec les outils de supervision et de gestion de parc, les problématiques de la localisation du traitement des données, et du transport des informations de trace, restent à être étudiées en fonction de la capacité des machines et de la confiance accordée au médium de transmission.

RÉFÉRENCES

ADAMS, K. et AGESEN, O. (2006). A comparison of software and hardware techniques for x86 virtualization. *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*. ACM, New York, NY, USA, ASPLOS-XII, 2–13.

AMSDEN, Z. (2010). *Timekeeping Virtualization for X86-Based Architectures*. Red Hat.

AVI KIVITY, YANIV KAMAY, D. L. U. L. A. L. (2007). kvm : the linux virtual machine monitor. *OLS '07 : The 2007 Ottawa Linux Symposium*.

BIEDERMAN, E. W. (2006). Multiple Instances of the Global Linux Namespaces. *Proceedings of Ottawa Linux Symposium 2006*.

BRYAN M. CANTRILL, M. W. S. et ADAM H. LEVENTHAL, S. M. (2004). Dynamic instrumentation of production systems. *Proceedings of USENIX 2004 Annual Technical Conference*. vol. Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal, Sun Microsystems.

CORBET, J. (2007). Introducing utrace. http ://lwn.net/Articles/224772/.

CORBET, J. (2009). Seccomp and sandboxing. http ://lwn.net/Articles/332974/.

DANIEL PRICE, A. T. (2004). Solaris zones : Operating system support for consolidating commercial workloads. *Proceedings of LISA '04 : Eighteenth Systems Administration Conference*.

DESNOYERS, M. (2009). *LOW-IMPACT OPERATING SYSTEM TRACING*. Thèse de doctorat, ECOLE POLYTECHNIQUE DE MONTREAL.

DESNOYERS, M. (2011). Common trace format (ctf) specification.

EDGE, J. (2009a). A look at ftrace. http ://lwn.net/Articles/322666/.

EDGE, J. (2009b). Perfcounters added to the mainline. http ://lwn.net/Articles/339361/.

GEORGE W. DUNLAP, SAMUEL T. KING, S. C. M. A. B. P. M. C. (2002). Revirt : Enabling intrusion analysis through virtual-machine logging and replay. *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation (OSDI)*.

GORMAN, M. (2009). *Notes on Analysing Behaviour Using Events and Tracepoints*. Red Hat.

HEIDARI, P., DESNOYERS, M. et DAGENAIS, M. (2008). Performance analysis of virtual machines through tracing. *Proc. Canadian Conf. Electrical and Computer Engineering CCECE 2008*. 000261–000266.

M. DESNOYERS, M. R. D. (2006). The lttng tracer : a low impact performance and behavior monitor for gnu/linux. *Proceedings of Ottawa Linux Symposium 2006*. 209–223.

M. JABBARIFAR, R. ROY, M. D. et SHAMELI-SENDI, A. (2011). Optimum off-line trace synchronization of computer clusters. *Proceedings of 2011 the High Performance Computing and Simulation conference*.

MENAGE, P. (2008). *Cgroups*. Google, BULL, Silicon Graphics.

PAUL BARHAM, BORIS DRAGOVIC, K. F. S. H. T. H. A. H. R. N. I. P. A. W. (2003). Xen and the art of virtualization. *Proceedings of the nineteenth ACM symposium on Operating systems principles*.

PIERRE-MARC FOURNIER, MATHIEU DESNOYERS, M. R. D. (2009). Combined tracing of the kernel and applications with lttng. *Proceedings of Ottawa Linux Symposium 2009*.

ROSTEDT, S. (2005). An introduction to kprobes. http ://lwn.net/Articles/132196/.

ROSTEDT, S. (2010). Using the trace_event macro. http ://lwn.net/Articles/379903/.

RUAN, Y. et PAI, V. (2004). Making the "box" transparent : System call performance as a first-class result. *In Proceedings of the USENIX 2004 Annual Technical Conference*.

RUSSELL, R. (2008). virtio : towards a de-facto standard for virtual i/o devices. *ACM SIGOPS Operating Systems Review - Research and developments in the Linux kernel*.

STRACHEY, C. (1959). Time sharing in large fast computers. *Proceedings of the International Conference on Information Processing*.

SUSHANT SHARMA, P. G. B. et MACCABE, A. B. (2005). A framework for analyzing linux system overheads on hpc applications. *In proceedings of LACSI Symposium 2005*.

VARA PRASAD, WILLIAM COHEN, F. C. E. M. H. J. K. B. C. (2005). Locating system problems using dynamic instrumentation. *Proceedings of Ottawa Linux Symposium 2005*.

ZENG, S. et HAO, Q. (2009). Network i/o path analysis in the kernel-based virtual machine environment through tracing. *Proc. 1st Int Information Science and Engineering (ICISE) Conf.* 2658–2661.

ZHANG, X. et DONG, Y. (2008). Optimizing xen vmm based on intel virtualization technology. *International Conference on Internet Computing in Science and Engineering*, 367–374.

Annexe A

SystemTap device seek

```
#! /usr/bin/env stap
#
# Copyright (C) 2010 Red Hat, Inc.
# By Dominic Duval, Red Hat Inc.
# dduval@redhat.com
#
# Keeps track of seeks on devices.
# Shows how to use hist_log.
#
# USAGE: stap deviceseeks.stp
#

global seeks, oldsec

probe ioblock.request {
    sec = sector
    seeks[devname] <<< sec - oldsec[devname]
    oldsec[devname] = sector
}

probe end {
    printf("\n")
    foreach ([devname] in seeks- limit 5) {
        printf("Device: %s\n", devname)
        println(@hist_log(seeks[devname]))
    }
}
```

Annexe B

Exemple Kprobes

```
/* kprobebio.c
   This is a simple module to get information about block
   io operations.
   Will Cohen
*/

#include <linux/module.h>
#include <linux/kprobes.h>
#include <linux/blkdev.h>

static int count_generic_make_request = 0;

static int inst_generic_make_request(struct kprobe *p,
 struct pt_regs *regs)
{
 ++count_generic_make_request;
 return 0;
}

/*For each probe you need to allocate a kprobe structure*/
static struct kprobe kp = {
 .pre_handler = inst_generic_make_request,
 .post_handler = NULL,
 .fault_handler = NULL,
 .addr = (kprobe_opcode_t *) generic_make_request,
};

int init_module(void)
{
```

```
 register_kprobe(&kp);
 printk("kprobe registered\n");
 return 0;
}

void cleanup_module(void)
{
 unregister_kprobe(&kp);
 printk("kprobe unregistered\n");
 printk("generic_make_request() called %d times.\n",
 count_generic_make_request);
}

MODULE_LICENSE("GPL");
```